Markt&Technik

Florian Matthes

Pascal mit dem C128

Pascal-Kurs · Tips & Tricks
Programmierhandbuch

Auf Diskette:
Pascal-Compiler/C 128

Pascal mit dem C 128

Florian Matthes

Pascal

mit dem C 128

Pascal-Kurs
Tips & Tricks
Programmier-
handbuch

Markt & Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Matthes, Florian:
Pascal mit dem C 128 : Pascal-Kurs, Tips & Tricks ; Programmierhandbuch / Florian Matthes. –
Haar bei München : Markt-und-Technik-Verlag, 1987.
ISBN 3-89090-386-X

Die Informationen im vorliegenden Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Buch gezeigten Modelle und Arbeiten ist nicht zulässig. »Commodore 64« und »Commodore 128 Personal Computer« sind Produktbezeichnungen der Commodore Büromaschinen GmbH, Frankfurt, die ebenso wie der Name »Commodore« Schutzrecht genießen. Der Gebrauch bzw. die Verwendung bedarf der Erlaubnis der Schutzrechtsinhaberin.

15 14 13 12 11 10 9 8 7 6 5 4 3 2
90 89 88 87

ISBN 3-89090-386-X

© 1987 by Markt&Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/West-Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Kösel, Kempten
Printed in Germany

Inhaltsverzeichnis

	Vorwort	9
1	**Die Werkzeuge**	11
1.1	Warum Pascal?	11
1.2	Arbeitsweise eines Compilers	12
1.3	Erste Schritte mit dem Pascal-System für den C128	14
1.3.1	Installation des Pascal-Systems	14
1.3.2	Start des Systems	16
1.3.3	Der Editor	17
1.3.4	Der Compiler Pascal 2.0	20
1.3.5	Das Laufzeitsystem	21
2	**Einführung in Pascal**	25
2.1	Symbole und Syntax-Diagramme	25
2.2	Allgemeine Programmstruktur	29
2.3	Deklaration von Variablen	31
2.4	Anweisungen und Ausdrücke	33
2.5	Einfache Ein- und Ausgabe am Bildschirm	35
2.5.1	WRITE	35
2.5.2	READ	38
2.6	Elementare Datentypen	42
2.6.1	Der Typ INTEGER	43
2.6.2	Der Typ REAL	44
2.6.3	Gegenüberstellung REAL und INTEGER	47
2.6.4	Der Typ CHAR	48
2.6.5	Der Typ BOOLEAN	50
2.6.6	Der Typ STRING	52
2.7	Deklaration von Konstanten	54
2.8	Kontrollstrukturen	55
2.8.1	Anweisungsfolgen	56
2.8.2	Bedingte Anweisungen	58

2.8.3	Fallunterscheidung	61
2.8.4	WHILE-Anweisung	64
2.8.5	REPEAT-Anweisung	69
2.8.6	FOR-Anweisung	72
2.8.7	Sprunganweisung	75
2.9	Die Datenstruktur Array	80
2.9.1	Eindimensionale Arrays	80
2.9.2	Stringoperationen	89
2.9.3	Mehrdimensionale Arrays	93
2.10	Deklaration von Typen	104
2.11	Prozeduren	107
2.11.1	Lokalität von Namen	109
2.11.2	Parameter für Prozeduren	112
2.11.3	Funktionen	116
2.11.4	Standardprozeduren	120
2.11.5	Rekursion	121
2.11.6	FORWARD-Deklarationen	129
2.12	Skalare Typen und ihre Operationen	135
2.12.1	Aufzählungstypen	135
2.12.2	Unterbereichstypen	139
2.13	Mengentypen	142
2.14	Der Datentyp Record	148
2.15	Variante Records	157
2.16	Der Datentyp File	163
2.16.1	Sequentiell schreiben	164
2.16.2	Sequentiell lesen	165
2.17	Textfiles	175
2.18	Dynamische Datenstrukturen	185
2.18.1	Lineare Strukturen (Listen)	191
2.18.2	Bäume	203
3	**Tips und Tricks**	211
3.1	Files im C128-Modus	211
3.2	Pascal und Maschinensprache auf dem C128	224
3.3	Nützliche Pascal-Routinen	229
3.4	Automatisches Nachladen von Programmteilen	234
4	**Dokumentation Pascal-System**	241
4.1	Das Pascal-Menü	243
4.2	Der Editor	245
4.2.1	Arbeitsweise des Editors	245

4.2.2	Gliederung des Bildschirms	245
4.2.2.1	Die Kopfzeile	246
4.2.2.2	Der Scroll-Betrag	247
4.2.2.3	Die Statuszeilen	247
4.2.2.4	Die Zeilen TOP und BOTTOM	248
4.2.2.5	Der Zeilennummernbereich	248
4.2.2.6	Das Textfenster	249
4.2.3	Cursorsteuerung	249
4.2.4	Die »primary-commands«	251
4.2.5	Die »line-commands«	255
4.2.6	Der Textmodus	258
4.2.7	FIND	259
4.2.8	CHANGE	262
4.2.9	Die Tasten <F2> und <F4>	263
4.2.10	INPUT	264
4.2.11	OUTPUT	266
4.2.12	COPY	268
4.2.13	Belegung eigener Funktionstasten	269
4.2.14	Meldungen innerhalb des Editors	271
4.3	Bedienung des Compilers Pascal 2.0	272
4.3.1	Wahl der Optionen	272
4.3.2	Meldungen innerhalb des Compilers	274
4.3.3	Rückkehr zu BASIC	276
4.3.4	Speicherverteilung im Pascal-System	279
4.4	Sprachbeschreibung Pascal 2.0	284
4.4.1	Grundsätzliches	284
4.4.2	Implementierungsabhängige Details	285
4.4.3	Reservierte Wortsymbole	296
4.4.4	Vordefinierte Namen	296
4.4.4.1	Namen für Konstanten	297
4.4.4.2	Namen für Typen	297
4.4.4.3	Namen für Variablen	297
4.4.4.4	Prozeduren für dynamische Objekte	298
4.4.4.5	Ein- und Ausgabeoperationen	300
4.4.4.6	Arithmetische Funktionen	306
4.4.4.7	Vordefinierte String-Prozeduren	308
4.4.4.8	Systemnahe vordefinierte Prozeduren	311
4.4.4.9	Vordefinierte Grafikprozeduren	314
4.4.4.10	Weitere vordefinierte Prozeduren	320
4.4.5	Files in Pascal 2.0	324
4.4.5.1	Übernahme von Programmen mit Files	326

4.4.6	Aktive Kommentare	328
4.4.6.1	Bereichsüberprüfung	329
4.4.6.2	Parameterüberprüfung	331
4.4.6.3	Chain-Option	332
4.4.6.4	Include-Files	333
	Anhang	
A	Syntax-Diagramme Pascal 2.0	335
B	Fehlernummern Pascal 2.0	341
C	Laufzeitfehler	347
D	Operatoren in Pascal 2.0	349
E	Literaturhinweise	351
	Stichwortverzeichnis	353
	Hinweise auf weitere Markt & Technik-Produkte	356

Vorwort

Das vorliegende Buch richtet sich an alle Besitzer eines C 128 oder C 128D, die eine moderne Programmiersprache auf ihrem Rechner einsetzen möchten. Obwohl die volle Leistungsfähigkeit der Sprache durch das auf Diskette mitgelieferte Pascal-System unterstützt wird, ist es aufgrund seiner einfachen Bedienung auch für den Anfänger geeignet.
Durch die Einheit von Buch und Diskette können die nicht zu unterschätzenden Schwierigkeiten eines Anfängers beim Übergang von einem interaktiven BASIC-System zu einer kompilierten Sprache ausgeräumt werden. Das erste Kapitel beschreibt daher zunächst an einem Beispiel Schritt für Schritt die Bedienung des Systems.
Kapitel 2 stellt einen vollständigen Pascal-Kurs dar. Es werden dort nur geringe Kenntnisse in der Programmierung vorausgesetzt. Da der beiliegende Compiler den üblichen Standardsprachumfang [1] akzeptiert, können alle Eigenschaften der Sprache in Beispielen vorgestellt werden. In einigen Programmbeispielen werden die Grafikoperationen von *Pascal 2.0*, die an BASIC 7.0 angelehnt sind, verwendet, um komplizierte Zusammenhänge (z.B. die Rekursion) anschaulich und leicht verständlich darzustellen.
Die Beispiele sollen soweit möglich nicht Selbstzweck sein, sondern auch als Vorlage zur Behandlung von Standardproblemen dienen. Alle wichtigen Routinen werden jeweils in Form eines vollständigen Programmes vorgestellt, damit sie im korrekten Kontext direkt ablauffähig sind. Schließlich werden im gesamten zweiten Kapitel Anregungen gegeben, das erworbene Wissen durch »Experimente« und die Lösung von Übungsaufgaben in der Praxis zu festigen.
Kapitel 3 richtet sich an den fortgeschrittenen Anwender. Hier wird zum Beispiel erläutert, wie man Pascal-Programme mit Assembler-Programmen koppelt. Außerdem werden rechnerspezifische Routinen in Pascal aufgeführt sowie Tips zum Pascal-System gegeben. In Kapitel 4 ist die Dokumentation des Pascal-Systems zusammengestellt. Sie gibt klare Auskunft auch über Details des Editors, des Compilers und die zahlreichen vordefinierten Prozeduren und Funktionen.

Kapitel 1
Die Werkzeuge

Auch wenn Sie bereits mit anderen Compilern gearbeitet haben, sollten Sie diesen ersten Teil nicht einfach überschlagen. Sie erhalten in diesem Kapitel nämlich einen Überblick über die Funktionsweise des gesamten Pascal-Systems, während in der eigentlichen Dokumentation in Kapitel 4 eher die Details des Editors und Compilers behandelt werden.

1.1 Warum Pascal?

Gegenüberstellung BASIC–Pascal

Während die Sprache BASIC praktisch zum Lieferumfang jedes Mikrocomputers gehört und jede Computerzeitschrift seitenweise Listings für die unterschiedlichsten BASIC-Dialekte abdruckt, ist der Kreis der Pascal-Benutzer gerade bei den Besitzern der sogenannten *Heimcomputer* noch recht klein.
Auf den ersten Blick scheint das Programmieren in Pascal auch eine recht unangenehme Sache zu sein: Zwischen der Eingabe eines Programmes und seiner Ausführung liegen zahlreiche Schritte, bei denen an den unmöglichsten Stellen Fehler gemeldet werden. Besonders unangenehm sind jedoch jene Fehler, die erst zur Programmlaufzeit auftreten. Mit einer kryptischen Meldung wird das Programm beendet, ohne daß man die Fehlerposition oder die Inhalte der Variablen kennt. Um nur einen einzigen Befehl zu korrigieren, muß man zu allem Überfluß anschließend auch noch das gesamte Programm neu übersetzen.
Der Vergleich mit BASIC bietet also ein recht abschreckendes Bild. Jedoch wurde bei dieser Gegenüberstellung die **Sprache** Pascal mit keinem Wort erwähnt. Die Gründe für den Einsatz von Pascal treten eben nicht beim Vergleich der Werkzeuge, sondern bei der Analyse der Leistungsfähigkeit der Sprachkonzepte zutage.

Analyse der Leistungsfähigkeit

Pascal als moderne Programmiersprache

Nicht umsonst ist Pascal **die** Ausbildungssprache zur Strukturierten Programmierung. Im Sprachumfang von Pascal sind alle Konzepte zur Gliederung von Anweisungen und Datenstrukturen klar und übersichtlich realisiert worden. Somit bietet *Pascal 2.0* auf dem C128 eine ideale Möglichkeit, die Sprache

Pascal als Beispiel für eine moderne Programmiersprache zu erlernen, um später dieses Wissen in Schule, Universität oder Beruf zu verwenden.
Andererseits sind auch die meisten der üblichen Anwendungsprogramme auf Mikrocomputern (Dateiprogramme, graphische Auswertungen, Mathematikprogramme, Logikspiele etc.) besser in Pascal als in BASIC zu realisieren. Außerdem sprechen eine höhere Codedichte und Ausführungsgeschwindigkeit bei großen Programmen für die Verwendung von Pascal.

Portabilität von Pascal-Programmen

Nicht zu vernachlässigen ist die Portabilität der Programme. Ein Programm, das auf einem C 128 in Pascal erstellt wurde und keine speziellen Eigenschaften des C 128 nutzt (Grafik- oder Systembefehle), kann direkt auf einen anderen Computer übernommen werden. In Kapitel 4.4.1 finden Sie einige Anmerkungen zur Kompatibilität zwischen den einzelnen Pascal-Implementationen.
Die meisten Compiler bieten jedoch auch computerspezifische Erweiterungen der Sprache Pascal in Form von zusätzlichen Typen oder vordeklarierten Prozeduren (»Befehlen«) an, die die Einsatzgebiete der Sprache Pascal deutlich erweitern.

Bessere Bedienerführung in Pascal-Compilern

In den letzten Jahren ist außerdem die Bedienung der Pascal-Compiler für Mikrocomputer deutlich verbessert worden. Während die »frühen« Implementationen sich noch stark an dem *Batch*-Betrieb der Fortran- und COBOL-Compiler orientierten, sind bei neueren Systemen Editor, Compiler und Laufzeitsystem stark verzahnt und erlauben einen zügigeren Entwicklungsprozeß.

Assemblerprogramme für laufzeitkritische Anwendungen

Neben diesen Vorteilen dürfen aber auch die Grenzen von Pascal nicht verschleiert werden. Auf Mikrocomputern existieren heutzutage nur wenige »Hochsprachen-Architekturen«, so daß Assemblerprogramme in laufzeitkritischen Anwendungen immer noch um ein oder zwei Zehnerpotenzen schneller sind. Außerdem ist die Entwicklung von Minimalprogrammen mit weniger als 100 Zeilen gerade bei ROM-residenten BASIC-Interpretern deutlich einfacher.

1.2 Arbeitsweise eines Compilers

Aufgaben eines Compilers

Um die Funktionsweise des Pascal-Systems zu verstehen, muß zunächst die grundsätzliche Aufgabe eines Compilers erläutert werden.
Vielleicht wissen Sie bereits, daß kein Mikrocomputer direkt BASIC oder Pascal versteht, sondern nur in seiner speziellen primitiven Maschinensprache programmiert werden kann. Andererseits können Sie ja offensichtlich den C 128 mit Befehlen wie PRINT 6*4 oder GOTO 9 zu sinnvollen Tätigkeiten bewegen. Darüber hinaus verspricht Ihnen dieses Buch, auch in Pascal mit dem C 128 kommunizieren zu können.

Übersetzung Hochsprache in Maschinensprache

Die Lösung dieses Dilemmas ist die Existenz von Hilfsprogrammen, die BASIC- oder Pascal-Befehle in eine Folge vieler Befehle in Maschinensprache übersetzen. Diese Übersetzungsprogramme selbst sind vollständig in der

Maschinensprache des Mikroprozessors (des 8502 im C 128) geschrieben und somit von diesem direkt ausführbar.

BASIC-Hilfsprogramme im Betriebssystem integriert

Beim C 128 befindet sich dieses Hilfsprogramm für BASIC bereits beim Einschalten im Rechner, da es zusammen mit dem Betriebssystem unlöschbar in sogenannten ROMs gespeichert ist. Wenn Sie in BASIC eine Zeile mit Zeilennummer eingeben, so wird diese Zeile im Programmspeicher des Rechners gespeichert. Beim Start des Programmes mit RUN wird das Programm Befehl für Befehl gelesen. Für jeden Befehl wird ein kleines Maschinensprache-Programm im ROM aufgerufen, das den jeweiligen BASIC-Befehl ausführt. Bei dem Befehl PRINT 6*3 z.B. würde zunächst eine Multiplikationsroutine und dann eine Ausgaberoutine gestartet. Falls Sie bei der Eingabe Fehler gemacht haben, meldet dies das System mit Angabe der Zeilennummer des fehlerhaften Befehls:

```
SYNTAX ERROR IN 312
```

Interpreter

Eine Programmiersprache, deren Programme nach diesem Schema ausgeführt werden, nennt man eine interpretierte Sprache. Das Hilfsprogramm im ROM heißt deshalb auch **Interpreter**. Durch diese interpretative Ausführung können Sie in BASIC beliebig zwischen Programmausführung und Programmänderung wechseln und sogar Befehle ohne Zeilennummern direkt ausführen.

Compiler

Das Pascal-System auf der beiliegenden Diskette enthält einen **Compiler**. Dies ist ein Programm, das ebenfalls eine Übersetzung der *höheren* (problemorientierten) Programmiersprache Pascal in einen für den Computer verständlichen Code vornimmt. Der Übersetzungsvorgang unterscheidet sich wie folgt von der Arbeitsweise eines Interpreters:

Übersetzungsvorgang beim Compiler

Zunächst erstellen Sie ein vollständiges(!) Programm in Pascal. Dieses Programm geben Sie mit einem **Editor**, also einem Textverarbeitungsprogramm, ein. Dieses Programm heißt **Quelltext** (engl. *source code*). Der Compiler liest diesen Programmtext in einem Durchlauf, wobei er prüft, ob das Programm den Syntaxregeln für Pascal entspricht. Eventuelle Fehler werden mit einem Hinweis auf die Art des Fehlers markiert. Gleichzeitig werden fehlerfreie Anweisungen in eine Folge von Befehlen in Maschinensprache übersetzt.

Ergebnis der Übersetzung ist also ein Programm, das vom Rechner ohne weitere Hilfsprogramme ausgeführt werden kann. Dieses Programm bezeichnet man als **Objektprogramm** (engl. *object code*). Theoretisch könnten Sie jetzt den Quelltext löschen, da dieser zur Ausführung des Objektprogrammes nicht mehr benötigt wird. Natürlich werden Sie das nicht tun, da das Programm noch logische Fehler enthalten kann, die der Compiler nicht entdeckt.

Laufzeitfehler

Obwohl ein Objektprogramm per Definition frei von syntaktischen Fehlern ist, können auch zur Laufzeit Fehler auftreten. Ein typisches Beispiel ist eine Division durch null. In diesem Fall läßt sich von dem Fehler im Objektprogramm nicht ohne weiteres auf den fehlerhaften Befehl im Pascal-Quelltext schließen.

Das mitgelieferte Pascal-System unterstützt Sie auch in diesem Fall, indem Sie den Compiler zum Lokalisieren von Laufzeitfehlern verwenden können.

Korrekturen von Fehlern

Zur Korrektur von logischen oder syntaktischen Fehlern müssen Sie wieder von vorn anfangen, da der Quelltext nach einer Änderung immer vollständig übersetzt werden muß.

Bei einigen Compilern liegen zwischen dem Quelltext und dem Objektprogramm noch mehrere Schritte, in denen weitere Hilfsprogramme (*linker*, *assembler*, *loader*) aufgerufen werden.

1.3 Erste Schritte mit dem Pascal-System für den C128

Erste Schritte zur Bedienung des Pascal-Systems

In diesem Kapitel werden noch keine Eigenschaften der Sprache Pascal vorgestellt, sondern nur die ersten Schritte bei der Bedienung des Pascal-Systems genau erklärt. Nachdem Sie dieses Kapitel bearbeitet haben, kennen Sie das Zusammenspiel der Komponenten des Systems, so daß Sie sich ohne Probleme in der Dokumentation in Kapitel 4 zurechtfinden werden.

1.3.1 Installation des Pascal-Systems

Anlegen einer System- und Arbeitsdiskette

Bei der Arbeit mit dem Pascal-System sollten Sie nicht die beiliegende **Originaldiskette**, sondern nur eine Kopie, die sogenannte **Systemdiskette**, zum Start des Pascal-Systems verwenden. Nach dem Start des Systems können Sie die Systemdiskette aus dem Diskettenlaufwerk nehmen und eine **Arbeitsdiskette** einlegen. Auf dieser Diskette können Sie Quelltexte und übersetzte Programme speichern. Außerdem benötigen übersetzte Programme die Datei PAS.LIB. Möchten Sie zu Programmfehlern eine ausführliche Fehlermeldung in englischer Sprache erhalten, müssen Sie die Datei ERRORS.TXT ebenfalls auf die Arbeitsdiskette kopieren.

Inhalt der **Originaldiskette**		Inhalt der **Systemdiskette**	
PASCAL	PRG	PASCAL	PRG
SYSTEM.M	PRG	SYSTEM.M	PRG
EDITOR.M	PRG	EDITOR.M	PRG
COMPILER.M	PRG	COMPILER.M	PRG
PAS.LIB	PRG	PAS.LIB	PRG
ERRORS.TXT	SEQ	ERRORS.TXT	SEQ
und weitere Beispielprogramme		und beliebige weitere Daten	
Inhalt der **Arbeitsdiskette**			
PAS.LIB	PRG	Laufzeitbibliothek für Objektprogramme	
ERRORS.TXT	SEQ	Fehlermeldungen (können entfallen)	
Quelltexte, Objektprogramme und beliebige weitere Dateien			

Um Ihnen das Erstellen der System- und Arbeitsdiskette zu erleichtern, können Sie **vor** dem Start des Pascal-Systems menügesteuert die einzelnen Dateien kopieren. In diesem Menü können Sie außerdem die Farben bei der Darstellung auf einem Fernsehgerät oder Monitor Ihren Bedürfnissen anpassen.

Laden des Pascal-Systems

Formatieren Sie zunächst eine Diskette, die Sie als Systemdiskette verwenden wollen. Entfernen Sie dann alle Erweiterungsmodule, schalten Sie den C 128 aus und dann wieder ein, um alle geladenen Maschinenprogramme zu löschen, und laden Sie anschließend das Pascal-System von der Originaldiskette:

```
LOAD"PASCAL",8
RUN
```

Nach einer Ladezeit, die stark vom verwendeten Floppy-Laufwerktyp (1541 oder 1571) abhängt, erscheint folgendes Auswahlmenü:

```
            PASCAL-SYSTEM
            =============
  1. System starten
  2. Fehlermeldungen kopieren
  3. "PAS.LIB" kopieren
  4. Systemdiskette kopieren
  5. System neu konfigurieren
  6. Abbruch (Ausgang zu BASIC)
 ==>1
```

Bild 1: Installationsmenü des Pascal-Systems

Optionen im Installations-Menü

Als Besitzer eines Farbmonitors oder Farbfernsehers können Sie die voreingestellten Farben des Pascal-Systems verwenden. Für Schwarzweiß-Bildschirme kann jedoch eine Umschaltung der Farben notwendig sein. Bevor Sie einen der Menüpunkte wählen, lesen Sie bitte zunächst die nachfolgende Beschreibung aller wählbaren Operationen.

1. Das bereits geladene Pascal-System wird gestartet. Eine Rückkehr in das Installationsmenü ist nicht mehr möglich. Nach dem Start des Pascal-Systems können Sie die Systemdiskette aus dem Diskettenlaufwerk entnehmen.

Kopieren der System-Files

2. Mit diesem Menüpunkt wird zunächst die Datei ERRORS.TXT von der Originaldiskette gelesen. Danach erscheint eine Aufforderung zum Einlegen der **Arbeitsdiskette**, auf die Sie die Fehlermeldungen kopieren möchten. Schließlich wird die Datei ERRORS.TXT auf dieser Diskette gespeichert.
3. Bei der Auswahl dieses Menüpunktes werden Sie zum Einlegen der Zieldiskette aufgefordert. Dann wird die bereits im Hauptspeicher vorhandene Datei PAS.LIB auf dieser Diskette gespeichert.

4. Zunächst wird die Datei ERRORS.TXT von der Originaldiskette gelesen. Anschließend werden Sie zum Einlegen der Zieldiskette aufgefordert. Auf dieser Diskette werden dann die Dateien PASCAL.P PRG, SYSTEM.M PRG, EDITOR.M PRG, COMPILER.M PRG, PAS.LIB PRG und ERRORS.TXT SEQ kopiert. Hatten Sie zuvor den Menüpunkt 5 gewählt, so werden die entsprechenden Farben fest auf der neuen Systemdiskette eingestellt.

Konfiguration der Farbdarstellung

5. **Vor** dem Erstellen einer Systemdiskette können Sie die Farbdarstellung für den 40- und 80-Zeichen-Modus getrennt wählen. In einem Menü können Sie durch wiederholte Betätigung der Zifferntasten <0> bis <9> die Farben innerhalb des Editors und des Compilers wählen. Jeder Tastendruck schaltet die entsprechende Farbe weiter. Jede Farbe kann aus einer Palette der folgenden Farben gewählt werden:
   schwarz, weiß, rot, cyan, violett, grün, blau, gelb, orange, braun, hellrot, grau 1, grau 2, hellgrün, hellblau und grau 3
   Auf dem 80-Zeichen-Bildschirm stehen folgende Farben zur Verfügung:
   schwarz, grau 1, blau, hellblau, grün, hellgrün, beige, zinnober, rot, hellrot, lila, violett, braun, gelb, hellgrau, weiß
6. Mit diesem Menüpunkt verlassen Sie das Installationsmenü und kehren zu BASIC zurück. Bei einem erneuten Start mit <RUN> werden wiederum alle Systemdateien geladen.

Im Installationsmenü wird die Auswahl »1« bereits vorgegeben, so daß Sie zum Start des Systems nur die Return-Taste betätigen müssen.

Anlegen einer Sicherungskopie

Beim ersten Start des Pascal-Systems lassen Sie bitte die Originaldiskette eingelegt und wählen Sie den Menüpunkt 4. Nun wird die Datei ERRORS.TXT von der Diskette gelesen. Danach erscheint folgende Meldung:

```
Bitte legen Sie jetzt die formatierte
Zieldiskette in das Diskettenlaufwerk.
Weiter mit einer beliebigen Taste
```

Nachdem Sie eine formatierte leere Diskette eingelegt haben, drücken Sie z.B. die Return-Taste. Nun werden alle benötigten Dateien auf die Leerdiskette kopiert. Diese Diskette wird in der Anleitung **Systemdiskette** genannt.
Möchten Sie bereits jetzt eine **Arbeitsdiskette** anlegen, so können Sie im Installationsmenü nacheinander die Punkte 2 und 3 wählen. Dadurch werden die Dateien PAS.LIB und ERRORS.TXT auf die Arbeitsdiskette kopiert.
Nun können Sie die Originaldiskette an einem sicheren Ort verwahren, um versehentlich zerstörte Systemdateien auf der **Systemdiskette** zu ersetzen.

1.3.2 Start des Systems

Nach den vorbereitenden Arbeiten des vorangegangenen Abschnittes können Sie sich jetzt der Bedienung des eigentlichen Pascal-Systems zuwenden.

Regeln für Tastatureingaben

Alle Eingaben im System sind so organisiert, daß Sie mit möglichst wenigen Zwischenschritten jede Funktion erreichen können. Dabei besitzen im allgemeinen die Zeichen »*« und »?« eine Sonderfunktion. Eingaben bei blinkendem Cursor müssen mit der Return- oder Enter-Taste des Ziffernblockes beendet werden. Grundsätzlich wird bei längeren Operationen (Laden, Speichern, Kompilieren) eine Abfrage der Run/Stop-Taste vorgenommen, so daß die Ausführung abgebrochen werden kann.

Wichtige Einstellungen für das gesamte System

Compiler, Editor und Laufzeitsystem unterstützen sowohl den 40- als auch den 80-Zeichen-Modus. Außerhalb des Editors können Sie mit der Tastenkombination <Esc> <X> zwischen den beiden Bildschirmen hin- und herschalten. Innerhalb des Editors gibt es einen speziellen Umschaltbefehl. Beim Start des Pascal-Systems im 80-Zeichen-Modus wird automatisch der FAST-Modus (2 MHz Taktfrequenz) eingeschaltet. Dadurch wird die Ausführungsgeschwindigkeit im Editor und Compiler praktisch verdoppelt. Dabei wird jedoch der 40-Zeichen-Bildschirm deaktiviert. Es steht Ihnen frei, nach Belieben mit den Befehlen SLOW und FAST in BASIC oder Pascal zwischen 1 und 2 MHz Taktfrequenz zu wechseln. Schließlich können Sie noch mit der ASCII/DIN-Taste bei Bedarf eine deutsche Tastaturbelegung wählen. Jedoch ändert sich auch die Anzeige einiger Sonderzeichen (z.B. erscheint ein @ als §) am Bildschirm. Alle Beschreibungen in diesem Buch beziehen sich auf die ASCII-Tastaturbelegung (Taste ASCII/DIN nicht gedrückt).

Nach dem Start des Systems erscheint das zentrale Menü des Pascal-Systems. Von dort können Sie den Editor, Compiler und den BASIC-Interpreter zur Ausführung der übersetzten Programme erreichen.

```
PASCAL-MENU
-----------
SELECT OPTION:
NAME    EDIT NEW DATASET
'?'     RESUME EDIT
'$'     COMPILE DATASET
'%'     SHOW DIRECTORY
'*'     EXIT TO BASIC
==>
```

Bild 2: Das Pascal-Menü

1.3.3 Der Editor

Um Pascal-Quelltexte einzugeben und zu verändern, enthält das System einen komfortablen *full screen editor*. Mit diesem Programm können Sie den Bildschirm wie ein Fenster in allen vier Richtungen über den Text verschieben und

in diesem Fenster direkt Änderungen vornehmen, ohne sich um Zeilennummern zu kümmern.

Anlegen eines neuen Quelltextes

Sie sollen zur Übung ein Beispielprogramm mit dem Namen PROGRAMM1.P erstellen. Deshalb geben Sie im Pascal-Menü hinter dem Pfeil den Programmnamen an:

```
==>PROGRAMM1.P
```

Nachdem Sie diese Eingabe mit der Return-Taste beendet haben, sucht der Editor diesen Quelltext auf der eingelegten Diskette. Da noch kein Text mit diesem Namen existiert, teilt Ihnen der Editor mit, daß er einen neuen Text anlegen wird.

```
THIS IS A NEW DATASET!
ENTER LINE LENGTH:
[RANGE: 1..136]
['*' OR '?' RETURN TO MENU]
==>
```

Sie müssen nun eine Zahl eingeben, die die maximale Länge einer Textzeile bestimmt. Gültige Werte liegen zwischen 1 und 136 Zeichen pro Zeile:

```
==>60
```

Mit dieser Eingabe darf keine Zeile Ihres Programmes länger als 60 Zeichen werden. Wenn Sie die Eingabe mit <Return> abschließen, erscheint das eigentliche Editor-Bild (der 80-Zeichen-Bildschirm ist entsprechend breiter):

```
1.COL:0000                    SCROLL:HALF
    **************** TOP ***************
    ************** BOTTOM **************
```

Bild 3: Der Editor-Bildschirm

Blättern im Quelltext

In der ersten Zeile wird die Nummer der ersten Textspalte auf dem Bildschirm ausgegeben. Rechts außen in dieser Zeile steht der Betrag, um den das Textfenster beim Blättern (*scrolling*) verschoben wird. HALF bedeutet jeweils eine halbe Bildschirmseite. Das Blättern selbst erfolgt mit den Funktionstasten:

<F1> verschiebt das Fenster nach oben
<F3> verschiebt das Fenster nach unten
<F5> verschiebt das Fenster nach links
<F7> verschiebt das Fenster nach rechts

Der Cursor wird nicht blinkend dargestellt und wie üblich mit den Cursortasten über den Bildschirm bewegt. Zunächst können Sie sich Spaltenmarkierungen über dem Text zur Orientierung einblenden lassen. Dazu geben Sie in der ersten Bildschirmzeile **über** den Text 1.COL:000 das Kommando COLUMNS ein:

```
COLUMNS                          SCROLL:HALF
    **************** TOP ***************
    *************** BOTTOM ***************
```

Bild 4: Eingabe eines primary-commands

Eingabe eines Editor-Kommandos

Der zuvor angezeigte Text verschwindet automatisch. Um innerhalb des Editors Kommandos auszuführen, müssen Sie die Shift- und die Return-Taste drücken. Dadurch erscheint oberhalb der Zeile TOP eine Zeile mit Spaltenmarkierungen.
Da der Textspeicher leer ist, stehen die Zeilen TOP und BOTTOM direkt untereinander. Diese Zeilen stehen immer vor der ersten und nach der letzten Textzeile. Nur zwischen diesen Zeilen können Sie Texte eingeben. Um nun das Beispielprogramm einzugeben, müssen Sie erst einige Leerzeilen erzeugen. Dies geschieht, indem Sie in der Zeile TOP ganz links (vor den Sternen) den Befehl I15 (als Abkürzung für insert 15) eingeben:

```
1.COL:0000                       SCROLL:HALF
COL=0----+----1----+----2----+----3----+
I15 **************** TOP ***************
    *************** BOTTOM ***************
```

Bild 5: Eingabe eines line-commands

Wenn Sie wieder <Shift> <Return> drücken, werden am Textanfang 15 Leerzeilen eingefügt. Jetzt steht das Fenster am Textende, so daß nur die Zeile BOTTOM sichtbar ist. Durch zweimalige Betätigung der Funktionstaste F1 können Sie zum Textanfang blättern. Die eigentliche Texteingabe erfolgt rechts von den weißen Zeilennummern. Dabei springt der Cursor bei der Betätigung der Return-Taste auf den nächsten Zeilenanfang. Geben Sie bitte folgendes Programm ein:

```
1.COL:0000                       SCROLL:HALF
COL=0----+----1----+----2----+----3----+
   **************** TOP ***************
0000 PROGRAM PROGRAMM1(OUTPUT)
0001
0002   VAR I,N : INTEGER;
0003       R   : REAL;
0004
0005 BEGIN
0006  WRITE('N=') READLN(N);
```

Bild 6: Das erste Pascal-Programm (Fortsetzung nächste Seite) ►

```
0007  FOR I:= N TO 2*N DO
0008    BEGIN
0009      R:= 1/I;
0010      WRITELN(I:3, R:15);
0011    END;
0012 END.
0013
0014
   ************** BOTTOM **************
```

Bild 6: Das erste Pascal-Programm

Verlassen des Editors

Wenn Sie sich vertippt haben, können Sie Korrekturen mit der Del-Taste vornehmen. Nach der Eingabe des Textes verlassen Sie den Editor folgendermaßen: Bewegen Sie den Cursor mit der Home-Taste in die erste Bildschirmzeile. Dort geben Sie das Kommando

```
END
```

ein. Bei der Bestätigung der Eingabe mit Shift- und Return-Taste wird Ihr Text auf der eingelegten Diskette gespeichert. Anschließend erreichen Sie wieder das Pascal-Menü.

1.3.4 Der Compiler Pascal 2.0

Aufruf des Compilers

Um den soeben erstellten Quelltext im Arbeitsspeicher des Computers zu übersetzen, geben Sie im Pascal-Menü ein Dollarzeichen »$« ein. Es meldet sich der Compiler mit einer Copyright-Meldung und folgender Auswahlmeldung:

```
PASCAL 2.0 (Rx.x)
-----------------
SELECT OPTION:
0    CHECK SYNTAX
1    GENERATE CODE
ELSE LOCATE ADDRESS
==>1
```

Den voreingestellten Übersetzungsmodus (1 für Code-Erzeugung) bestätigen Sie einfach mit der Return-Taste.

```
LISTING TO PRINTER?
==>N
```

Durch die Betätigung der Return-Taste wird kein Listing am Drucker ausgegeben. Während der Kompilation wird der Quelltext am Bildschirm angezeigt. Sollten Sie sich bei der Eingabe des Programmes vertippt haben, markiert der Compiler den entsprechenden Fehler mit einem Pfeil. Durch die Eingabe eines

Sterns (»*«) brechen Sie dann die Übersetzung ab. Drücken Sie nur <Return>, so wird die Übersetzung fortgesetzt.
Traten keine Fehler auf, so erscheint die folgende Meldung:

```
ERRORS DETECTED: 0
     P-CODE FROM  7391 TO  7491
  CONSTANTS FROM  8192 TO  8195
(HIT RETURN FOR MENU)
```

Mit der Return-Taste kehren Sie zum Pascal-Menü zurück.

Beseitigen von Fehlern im Quelltext

Im Falle von Übersetzungsfehlern müssen Sie den Editor erneut aufrufen. Im Pascal-Menü geben Sie deshalb den Namen des Quelltextes ein:

```
==>PROGRAMM1.P
```

Wie im letzten Abschnitt erklärt, können Sie nun mit dem Editor Korrekturen vornehmen. Nachdem Sie die Korrekturen mit END gespeichert haben, können Sie vom Pascal-Menü aus den Compiler aufrufen.

1.3.5 Das Laufzeitsystem

Ist das Programm endlich fehlerfrei, so möchten Sie sicher als Lohn Ihrer Arbeit das übersetzte Programm auch einmal ausführen. Dazu verlassen Sie das Pascal-Menü mit der Eingabe »*« (EXIT TO BASIC). Es erscheint die übliche Einschaltmeldung von BASIC.

Das Objektprogramm steht im Arbeitsspeicher

Das Objektprogramm steht jetzt im Speicher. Es kann mit RUN, SAVE, VERIFY, LOAD, DSAVE, DVERIFY und DLOAD wie ein BASIC-Programm gestartet, gespeichert, geprüft und geladen werden. Sie dürfen jedoch keine Zeilen löschen oder hinzufügen, da sonst das Pascal-System nicht mehr korrekt arbeitet.
Der Objektcode wird durch eine BASIC-Zeile eingeleitet, die Sie mit LIST anzeigen können:

```
1986 GRAPHICCLR:BANK0:SYS7200: PASCAL 2.0
```

Nach dem Start mit RUN wird vom Beispielprogramm die Eingabe einer ganzen Zahl N erwartet. Anschließend werden alle Zahlen zwischen N und 2*N mit ihrem Kehrwert ausgedruckt:

Ausführung des Beispielprogramms

```
RUN
N=5
  5 2.00000000E-01
  6 1.66666667E-01
  7 1.42857143E-01
  8 1.11111111E-01
 10 1.00000000E-01
```

Bekanntermaßen besitzt jedes Programm einen Fehler, der erst bei der Programmausführung entdeckt wird. Den Fehler in PROGRAMM1 werden Sie bei der Eingabe von N=0 feststellen:

```
RUN
N=0
DIVISION BY ZERO
 ERROR AT 7458 IN PROGRAMM1
```

Die (dezimale) Zahl 7458 bezeichnet eine Adrese im Objektprogramm, bei der eine Division durch null stattgefunden hat. Zwar läßt sich in Ihrem ersten Trivialprogramm der fehlerhafte Divisionsbefehl ohne Mühe lokalisieren, jedoch kann in größeren Programmen die Fehlersuche eine recht aufwendige Angelegenheit sein.

Lokalisierung von Laufzeitfehlern im Quelltext

Daher bietet Ihnen der Compiler *Pascal 2.0* die Möglichkeit, zu Fehleradressen in einem Objektprogramm die zugehörige **Quelltext**position anzuzeigen. Um den Compiler erneut zu aktivieren, müssen Sie zum zentralen Pascal-Menü zurückkehren. Dies geschieht von BASIC aus durch die Eingabe eines Sterns (*) am Anfang einer Zeile (<Return> nicht vergessen!). Nach dem Aufruf des Compilers (»$« im Pascal-Menü) geben Sie die vom Laufzeitsystem angezeigte Fehleradresse ein:

```
PASCAL 2.0 (Rx.x)
-----------------
SELECT OPTION:
0    CHECK SYNTAX
1    GENERATE CODE
ELSE LOCATE ADDRESS
==>7458
```

Der Compiler durchläuft nun so lange den Quelltext, bis die Fehleradresse gefunden wurde. Unterhalb von Zeile 9 erscheint dann folgende Meldung:

```
        R:= 1 / I;
                ↑ ERROR 0 IN 9
'*' STOP  '?' EXPLANATION
ELSE CONTINUE
==>
```

Bei der Eingabe eines Fragezeichens (?) sucht der Compiler in der Datei ERRORS.TXT einen Text zu der Fehlernummer 0 und zeigt diesen an:

```
ADDRESS OF RUNTIME ERROR FOUND
```

Fehlermeldung im Klartext

Der Pfeil (↑) steht also auf der Position, bei der die illegale Division durch null auftrat. Im Anhang B sind alle Fehlernummern mit Erklärungen in deutscher Sprache aufgelistet.

Bild 7 zeigt noch einmal zusammenfassend die Komponenten des Pascal-Systems.

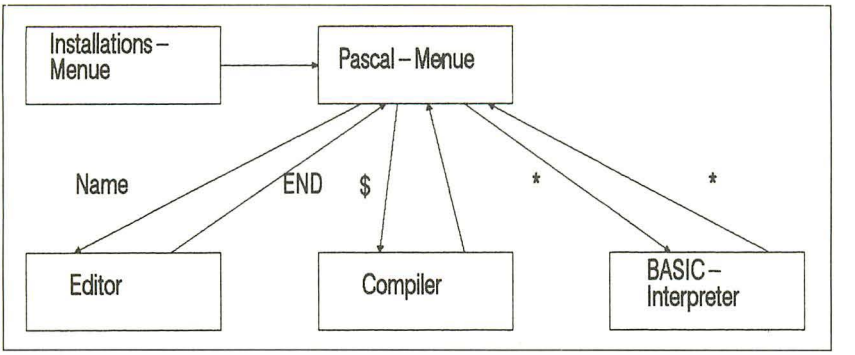


Bild 7: Struktur des Pascal-Systems

Anlegen von Sicherungskopien der Quelltexte

Zur Übung können Sie jetzt Sicherheitskopien der Pascal-Programme, die sich auf der **Originaldiskette** befinden, anlegen. Durch die Angabe des jeweiligen Programmnamens (mit dem Suffix .P oder .INC) laden Sie die Quelltexte in den Arbeitsspeicher des Editors, wechseln die Diskette und speichern anschließend die Texte mit END (im Editor) auf der neuen Diskette.

Aufgaben

Jetzt sollten Sie ein wenig in den Kapiteln 4.1 bis 4.3 der Dokumentation lesen. Dann werden Sie auch wissen, wie Sie aus dem Editor zurückkehren können, ohne daß der Quelltext gespeichert wird, was die Eingabe eines Fragezeichens im Pascal-Menü bewirkt und wie Sie das Listing von PROGRAMM1 auf den Drucker ausgeben können.

Möchten Sie den Compiler noch ein wenig beschäftigen, können Sie versuchen, einige der Beispielprogramme (Texte mit dem Suffix .P, aber *nicht* .INC) von der Originaldiskette zu übersetzen.

Kapitel 2
Einführung in Pascal

Ziel und Umfang des Kurses

Dieses Kapitel bietet einen vollständigen Einführungskurs in die Sprache Pascal. Nachdem Sie das gesamte zweite Kapitel bearbeitet und dabei Programmier-Erfahrung durch das Lösen der in den Text eingestreuten Aufgaben gesammelt haben, können Sie sich guten Gewissens einem der vielen Bücher für fortgeschrittene Pascal-Anwender zuwenden. Dabei möchte ich Sie bereits an dieser Stelle auf die Literaturhinweise in Anhang E aufmerksam machen. In vielen der dort genannten Bücher finden Sie Anregungen für interessante Programme, die wegen ihrer Komplexität den Rahmen dieses Einführungskurses sprengen würden.
Teilweise bietet *Pascal 2.0* Möglichkeiten, die über den *Standardsprachumfang*, wie er in der Definition der Sprache (dem sogenannten *report* [1]) festgelegt wurde, hinausgehen. Diese Erweiterungen werden jedoch an der jeweiligen Stelle angesprochen, so daß Sie beim Verzicht auf diese Erweiterungen Ihre Programme auch für andere Compiler und Computer verwenden können. Eine kurze Einführung zum Thema *Standardsprachumfang* findet sich in Abschnitt 4.4.1.

2.1 Symbole und Syntax-Diagramme

Leider liegt am Anfang Ihrer Arbeit mit Pascal eine *Durststrecke* von einigen Kapiteln, die sich mit etwas abstrakteren Grundlagen beschäftigen. Sollten Sie beim ersten Lesen einige Details nicht ganz verstehen, können Sie später, wenn Sie etwas praktische Erfahrung am Computer besitzen, diese Teile noch einmal bearbeiten.
Pascal ist eine **formale Sprache**: Programme sind nichts anderes als Symbolfolgen. Man kann zwar unendlich viele korrekte Folgen von Symbolen bilden, jedoch ist die Menge der Bildungsregeln für korrekte Programme, die **Syntax** der Sprache Pascal, endlich.

Symbole als Bausteine einer formalen Sprache

Dieses Kapitel beschäftigt sich mit den kleinsten Einheiten in einem Programmtext. Diese **Symbole** der Sprache Pascal lassen sich in folgende Gruppen einteilen, die nachfolgend im einzelnen vorgestellt werden:

- Namen (auch Bezeichner genannt)
- Zahlen
- Stringkonstanten
- Sonderzeichen
- Wortsymbole
- Kommentare

Die Definition einer Sprache über sozusagen *atomare* Symbole erlaubt eine gewisse Unabhängigkeit von den Eigenschaften spezieller Rechner. So wird z.B. in der Sprache nie Bezug auf Zeilennummern oder die Formatierung des Quelltextes genommen.

Regeln für Namen

Namen (Bezeichner) können vom Programmierer zur Identifikation der verschiedensten Objekte in einem Programm verwendet werden (Näheres in späteren Kapiteln). Formal gesehen besteht ein Name aus einem Buchstaben, gefolgt von Buchstaben oder Ziffern (»A« bis »Z« und »0« bis »9«). Ein Bezeichner kann also beliebig lang werden.

```
VARIABLE   B747   ERGEBNIS   A   MARION   B   JB007
```

sind also gültige Namen. Die folgenden Zeichenfolgen sind keine Namen:

```
3MAL      ERGEBNIS-3      MIT_UNDERSCORE
```

In *Pascal 2.0* sind nur die ersten 14 Zeichen eines Namens signifikant. Somit betrachtet der Compiler die folgenden Namen als gleich:

```
EXTRALANGERNAME1          EXTRALANGERNAME2
```

Eine Vielzahl von Namen besitzt in Pascal bereits eine vordefinierte Bedeutung, die in späteren Abschnitten besprochen wird.

Zahlen sind Symbole, die aus komplizierteren Zeichenfolgen bestehen können. Deshalb sind die Bildungsregeln auch nur schwerfällig in Worten zu beschreiben. Eine anschauliche und übersichtliche Beschreibung der Syntax von Pascal liefern die sogenannten **Syntaxdiagramme**. Bild 8 zeigt die Syntaxdiagramme für Zahlen in Pascal:

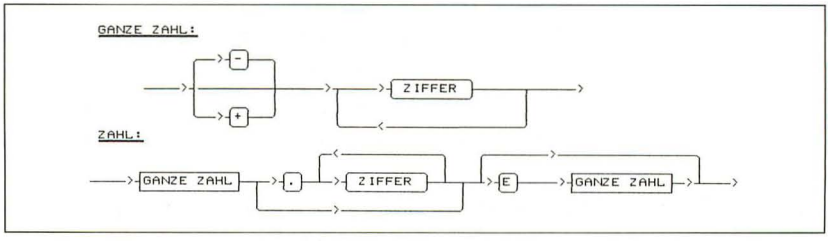


Bild 8: Zwei Syntaxdiagramme für Zahlen

So liest man Syntaxdiagramme

Indem man von links nach rechts den Pfeilen durch den Graphen folgt und die Zeichen in den Kästen mit den abgerundeten Ecken notiert, erhält man gültige Zahlen in Pascal. Dabei kann man an einer Verzweigung jeden beliebigen Weg wählen und sich auch in Schleifen bewegen. Es ist jedoch verboten, gegen die Pfeilrichtung zu laufen.

Eine *ganze Zahl* besteht also aus einer oder mehreren Ziffern. Außerdem kann der *Zahl* ein Vorzeichen vorausgehen. Im zweiten Diagramm mit dem Namen Zahl tritt zweimal ein Kästchen mit der Bezeichnung *ganze Zahl* auf. Da die Ecken dieses Kästchens nicht abgerundet sind, bedeutet dies, daß an dieser Stelle eine Zeichenfolge steht, die durch das erste Syntaxdiagramm *ganze Zahl* beschrieben wird.
In späteren Kapiteln sollen Sie diese Diagramme selbständig *lesen* können, um die Syntax von Anweisungen und Ausdrücken exakt zu verstehen. Daher sind alle Syntaxdiagramme der Sprache *Pascal 2.0* im Anhang A zusammengestellt.
Für Zahlen wird die Syntax an dieser Stelle noch einmal verbal beschrieben, um das Prinzip, das hinter den Diagrammen steht, zu verdeutlichen:
Eine *ganze Zahl* ist eine Ziffernfolge, eventuell mit Vorzeichen.

Regeln für Zahlen

Eine *Zahl* besteht aus einer *ganzen Zahl*. Daran kann sich ein Dezimalpunkt mit mindestens einer Nachkommastelle anschließen. Danach folgt eventuell der Buchstabe »E« mit einem Exponent. Dieser Exponent besteht ebenfalls aus einer *ganzen Zahl*.
Beispiele für zulässige Zahlen sind

```
1   0   1986   0.1   +22.3   -1E-4   1.5E8
```

Typische fehlerhafte Zahlen sind:

```
1.            (nach dem Punkt muß eine Ziffer folgen)
.1            (es muß eine Null vor dem Punkt stehen)
3,4           (ein Komma ist nicht zulässig)
3,000,000.00  (auch nicht hier)
```

In Pascal unterscheidet man also zwei **Typen** von Zahlen: Es gibt reelle und ganze Zahlen. Reelle Zahlen sind dadurch gekennzeichnet, daß sie Nachkommastellen und/oder einen Skalierungsfaktor (Exponent) besitzen. Der Skalierungsfaktor gibt an, um wie viele Stellen der Dezimalpunkt verschoben wird. Daher bezeichnen die folgenden Ziffernfolgen den gleichen Wert:

```
1.0 = 1E0 = 10E-1 = 100E-2 = 0.1E+1 = 0.01E+2
```

Eine **Stringkonstante** besteht in *Pascal 2.0* aus einer Folge von Zeichen, die in Hochkommata oder Anführungszeichen eingeschlossen ist.

Strings in Pascal 2.0

```
'Dies ist eine Stringkonstante'
"Dies ist ebenfalls ein String"
''
'Sonderzeichen: "!$%&/(?'
"klappt's?"
```

Um ein Anführungszeichen in einem String anzugeben, muß man den String mit Hochkommata einschließen. Außerdem können in *Pascal 2.0* nicht druckbare Sonderzeichen in eine Stringkonstante aufgenommen werden, indem man den dezimalen ASCII-Code des Zeichens hinter dem Zeichen » # « angibt. Das Klingelzeichen (bell) hat z.B. den ASCII-Code 7:

```
'Telefon ' #7 #7 #7 " klingelt!"
```

Stringkonstanten, die man so hintereinanderstellt, werden in *Pascal 2.0* zu einem String zusammengefaßt. Damit können insbesondere Strings definiert werden, die länger als eine Quelltextzeile sind. Das im *report* definierte Standard-Pascal ist in bezug auf Stringkonstanten restriktiver: Dort muß ein String mit Hochkommata eingeschlossen werden. Des weiteren darf ein String nicht leer sein, das Zeilenende nicht überschreiten und keine Kontrollzeichen enthalten. Zur Nennung eines Hochkommas muß dieses im String doppelt vorhanden sein.

Unterschiede zum report

In Pascal werden folgende **Sonderzeichen** verwendet:

Zusammengesetzte Symbole

+	Addition, Vereinigung von Mengen
–	Subtraktion, Differenz von Mengen
*	Multiplikation, Schnitt von Mengen
/	Division
:=	Zuweisung
=	gleich
<>	ungleich
>=	größer oder gleich
<=	kleiner oder gleich
( )	Klammern
[ ]	Index- und Mengenklammern
(* *)	Kommentarklammern
↑	Pfeil (Dereferenzieroperator)
..	Auslassungspunkte
, . ; :	Satzzeichen

Einige Symbole in der Liste bestehen aus zwei Sonderzeichen. Zwischen den beiden Zeichen darf dann kein Leerzeichen stehen:

:=	(dies ist ein Symbol)
: =	(dies sind zwei Symbole)

Reservierte Namen

Die reservierten **Wortsymbole** von Pascal sind in der folgenden Liste aufgeführt. Sie dürfen von Ihnen nicht als Namen verwendet werden. Ihre Bedeutung wird in den weiteren Kapiteln erklärt:

AND	FILE	NOT	TO
ARRAY	FOR	OF	TYPE
BEGIN	FORWARD	OR	UNTIL
CASE	FUNCTION	PACKED	VAR
CONST	GOTO	PROCEDURE	WHILE
DIV	IF	PROGRAM	WITH
DO	IN	RECORD	
DOWNTO	LABEL	REPEAT	
ELSE	MOD	SET	
END	NIL	THEN	

Im Gegensatz zu BASIC dürfen Namen Wortsymbole enthalten:

```
FORMEL     EINGABEENDE
```

sind also gültige Namen, obwohl sie die Zeichenfolgen FOR, OR, IN und END enthalten.

Tips zum Layout eines Programmes

Da die Größe eines übersetzten Programmes (Objektprogramm) nicht von der Formatierung des Quelltextes abhängig ist, spart man nicht wie in BASIC mit Leerzeichen zwischen den Symbolen. Vielmehr versucht man durch das Layout (Einrückung, Kommentare, sinnvolle Variablennamen) die Struktur des Programmes zu unterstreichen. Leerzeichen sind jedoch nur dann syntaktisch erforderlich, wenn durch ihr Fehlen aus zwei Symbolen eines würde:

```
IF X = 6 * Y THEN
```

In diesem Beispiel sind nur zwischen IF und X sowie zwischen Y und THEN Leerzeichen notwendig.

Kommentare sind überall möglich

Kommentare können an jeder Stelle des Programmes eingefügt werden, an der auch ein Leerzeichen stehen darf. Kommentare können beliebige Texte enthalten. Da auf dem C 128 keine geschweiften Klammern (»{« und »}«) darstellbar sind, werden Kommentarklammern durch »(*« und »*)« gebildet:

```
(* Dies ist ein Kommentar, der sich über mehrere Zeilen erstreckt *)
```

Wie dieses Beispiel zeigt, können Kommentare auch länger als eine Quelltextzeile werden. Natürlich darf ein Kommentar nicht die schließende Klammer »*)« enthalten.

Viele Compiler kennen auch **aktive Kommentare**, die den Kompilationsvorgang steuern. In *Pascal 2.0* beginnt ein aktiver Kommentar mit einem Dollarzeichen. Die Wirkung aller aktiven Kommentare ist in der Dokumentation beschrieben.

```
(*$R+  Code für Bereichstests erzeugen *)
```

2.2 Allgemeine Programmstruktur

Nachdem Sie im vorangegangenen Abschnitt die Symbole der Sprache Pascal als kleinste Elemente eines Programmes kennengelernt haben, wird in diesem Abschnitt die globale Struktur eines Pascal-Programmes vorgestellt. In Pascal genügt es nämlich nicht, die Befehle des Programmes einfach hintereinanderzustellen. Vielmehr ist – bildlich gesprochen – ein *Rahmen* erforderlich, der die eigentlichen Anweisungen umgibt.

Die Grobstruktur **jedes** Programmes läßt sich schematisch so angeben wie auf Seite 30 abgebildet.

Das Skelett eines Programmes

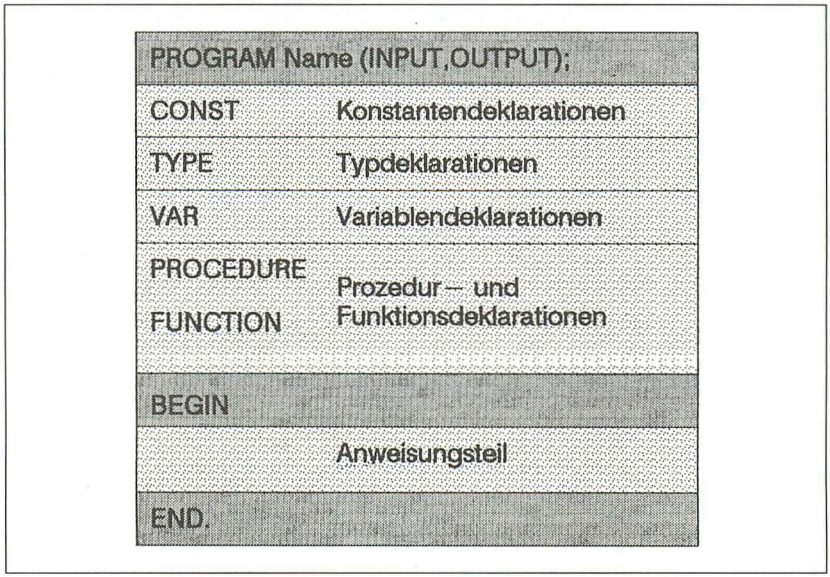


Bild 9: Programmstruktur in Pascal

Jeder der hell unterlegten Teile kann entfallen. Diese Teile werden nach und nach in den folgenden Abschnitten vorgestellt. Die dunkler unterlegten Teile müssen jedoch vorhanden sein. Das kürzeste korrekte Pascal-Programm, das keine Anweisung beinhaltet, sieht demnach folgendermaßen aus:

```
PROGRAM STRUKTUR(INPUT,OUTPUT);
BEGIN
END.
```

Die erste Zeile ist der **Programmkopf**. Hier wird nach dem Wortsymbol PROGRAM dem Programm ein Name gegeben, der jedoch im restlichen Programmtext keine weitere Bedeutung hat. Die Namen INPUT und OUTPUT deuten an, daß das Programm zwei *Kanäle* zur Umwelt besitzt: Dies ist die Tastatur als Standardeingabe und der Bildschirm als Standardausgabe. In Abschnitt 2.16 über Files wird auf diese Programmparameter näher eingegangen.

Namen müssen vor ihrer Verwendung deklariert werden

Bereits in dem in Kapitel 1 vorgestellten Beispielprogramm wurde der Variablenvereinbarungsteil benutzt, um die Variablen I, N und R zu deklarieren. Schon jetzt können Sie sich merken, daß in Pascal **alle** Namen deklariert werden müssen, bevor sie in Anweisungen verwendet werden können.

Bitte achten Sie insbesondere auf die Satzzeichen (Kommata, Punkte, Semikola). Sie sind genauso wichtig wie alle anderen Symbole und dürfen nicht fehlen.

Natürlich existiert auch ein Syntaxdiagramm, das diesen Aufbau eines Programmes definiert. Es heißt PROGRAMM und steht am Ende von Anhang A.

Wenn Sie das Programm in Bild 9 mit diesem Diagramm vergleichen, werden Sie den Pfeilen folgend zu dem Kasten BLOCK gelangen. Dieser bezieht sich auf das davorliegende Syntaxdiagramm BLOCK. Jeder Weg im Diagramm BLOCK führt vom Eingang links oben zum Ausgang rechts unten über die Symbole BEGIN und END. Auch wenn Sie die vielen Namen in Bild 9 und dem Syntaxdiagramm BLOCK noch nicht kennen, ist Ihnen sicher klargeworden, daß durch beide Darstellungsformen dieselben Regeln für den *Rahmen* eines Programmes in Pascal definiert werden.
Wenn Sie jetzt durch das Buch blättern, werden Sie feststellen, daß alle abgedruckten Listings diesem Schema gehorchen, selbst wenn die Deklarationsteile und der Anweisungsteil extrem lang werden oder leer sind.

Aufgaben

1. Das Programm STRUKTUR ist ein vollständiges Programm. Lassen Sie es deshalb einmal vom Compiler übersetzen. Experimentieren Sie ein wenig: Prüfen Sie am Programmbezeichner »STRUKTUR« die Regeln für Namen, indem Sie probeweise die Namen »STRUKTUR1« »2.PROGRAMM« »PROGRAM« etc. wählen. Entfernen Sie ein paar Satzzeichen, oder fügen Sie Teile des Beispielprogrammes aus Kapitel 1 ein. Sie sollten auch versuchen, das Programm in einer Zeile unterzubringen und Leerstellen einzufügen und zu löschen.
   Welche Fehlermeldungen liefert der Compiler?
2. An welcher Stelle im Syntaxdiagramm PROGRAMM kommt es zu Problemen, wenn Sie folgendes Programm untersuchen?
   `PROGRAM FALSCH (); BEGIN END.`
   Beheben Sie den Fehler!

2.3 Deklaration von Variablen

Eine Variable läßt sich unter zwei verschiedenen Gesichtspunkten betrachten. Einerseits dient sie bei der Programmlaufzeit zur Speicherung von veränderlichen (*variablen*) Werten, wie Zwischenergebnissen oder Zuständen des Programmes. Andererseits besitzt sie konstante Eigenschaften: Eine Variable wird über einen festen Namen (Variablenbezeichner) identifiziert. Außerdem kann sie nur eine gewisse Klasse von Werten speichern (z.B. nur Zeichen oder nur Zahlen).

Konstante Eigenschaften einer Variablen

Diese konstanten Eigenschaften werden im Vereinbarungsteil eines Pascal-Programmes für **jede** im Anweisungsteil benutzte Variable festgelegt. Sinnvollerweise wird man Variablen einen Namen geben, der ihre Bedeutung im Programm widerspiegelt:

Der Name einer Variablen

```
PROGRAM VARIABLENDEKLARATION(OUTPUT);
VAR  I          : INTEGER;
     ZAEHLER    : INTEGER;
     GEHALT     : REAL;
     DELTA      : REAL;
     GEFUNDEN   : BOOLEAN;
     ALTER      : INTEGER;
     BUCHSTABE1: CHAR;
     BEFEHL     : CHAR;
     ENDE       : BOOLEAN;
BEGIN END.
```

Eine Variablendeklaration beginnt mit dem reservierten Wortsymbol VAR. Anschließend werden die Namen aller im Programm verwendeten Variablen aufgeführt. Für jede Variable wird nach einem Doppelpunkt ihr Typ angegeben. Der **Typ** definiert die oben erwähnte Klasse von Werten, die eine Variable annehmen darf.

Der Typ einer Variablen

In diesem Beispiel werden die Typen durch vordefinierte Typnamen angegeben. An dieser Stelle sei nur soviel gesagt, daß die Variablen I, ZAEHLER und ALTER nur ganze Zahlen, GEHALT und DELTA reelle Zahlen, BUCHSTABE1 und BEFEHL jeweils nur ein Zeichen speichern können. Die Variablen GEFUNDEN und ENDE speichern hingegen Wahrheitswerte.

Die obige Variablendeklaration kann auch wie folgt abgekürzt werden:

```
VAR I, ZAEHLER, ALTER  : INTEGER;
    GEHALT, DELTA      : REAL;
    BUCHSTABE1, BEFEHL : CHAR;
    GEFUNDEN, ENDE     : BOOLEAN;
```

Um Mißverständnissen vorzubeugen, sei gesagt, daß die gewählte Anordnung der Deklarationen keine Bedeutung für den Compiler besitzt. Dieser beachtet nur die Reihenfolge der Symbole und nicht das Layout des Programmes, so daß folgender Quelltext ebenfalls korrekt ist

```
VAR  I,ZAEHLER,ALTER:INTEGER;GEHALT,DELTA:REAL;BUCHSTABE1,
BEFEHL:CHAR;GEFUNDEN,ENDE:BOOLEAN;
```

Jedoch sollte sich auch ein Mensch in Ihren Programmen zurechtfinden, für den Einrückungen die Übersicht sicher stark erhöhen.

Typische Fehler bei der Arbeit mit Variablen

Der entscheidende Vorteil einer expliziten Deklaration jeder Variablen am Programmanfang ist neben einer Dokumentation des Programmes die Möglichkeit, schon während der Übersetzung die Korrektheit von Operationen zu prüfen. Die folgende Zuweisung, die einer Variablen für ganze Zahlen ein Zeichen zuordnen würde, kann sofort vom Compiler als fehlerhaft erkannt werden:

```
PROGRAM TYPFEHLER (INPUT,OUTPUT);
  VAR ZAEHLER: INTEGER;
      BEFEHL : CHAR;
BEGIN
  ZAEHLER:= BEFEHL;
END.
```

Die Zuweisung ZAEHLER:= BEFEHL wird vom Compiler mit dem folgenden Fehlertext markiert:

```
144 ILLEGAL TYPE OF EXPRESSION
```

Versuchen Sie, im Programm eine undeklarierte Variable zu verwenden, so erhalten Sie vom Compiler die Fehlermeldung

```
59  ERROR IN VARIABLE (VARIABLE IDENTIFIER EXPECTED)
```

Während in BASIC jede Variable zum Programmbeginn mit dem Wert 0 bzw. dem leeren String vorbelegt wird, sind die Werte **aller** Variablen in Pascal zum Beginn der Programmausführung undefiniert.

2.4 Anweisungen und Ausdrücke

Nun haben Sie endlich das Rüstzeug beisammen, um sich dem Anweisungsteil des Programmes zuzuwenden. Der Anweisungsteil besteht aus einer (wie Sie gesehen haben eventuell sogar leeren) Folge von **Anweisungen** zwischen den Wortsymbolen BEGIN und END.

```
PROGRAM ANWEISUNGSFOLGE(INPUT,OUTPUT);
BEGIN
  WRITE('Diese Texte ');
  WRITE('werden in ');
  WRITE('der Reihenfolge ');
  WRITE('ihres Auftretens ');
  WRITE('gedruckt.')
END.
```

Das Semikolon trennt Anweisungen

In diesem Beispiel besteht jede Anweisung aus einem Befehl (genauer gesagt **Prozeduraufruf**) zur Ausgabe einer Stringkonstanten. Wenn Sie dieses Programm übersetzen und starten, sehen Sie, daß die Ausgaben in der Reihenfolge, in der sie im Programm genannt werden, ausgeführt werden. Die Anweisungen des Anweisungsteils sind voneinander durch Semikola getrennt. Vor dem Wortsymbol END kann das Semikolon entfallen.

Zuweisung von Ausdrücken an Variable

In diesem Abschnitt wird die elementarste Form der Anweisung, die **Zuweisung**, vorgestellt. Dabei wird einer (deklarierten!) Variablen links vom Zuweisungsoperator »:=« das Ergebnis der Berechnung des Ausdruckes auf der rechten Seite zugewiesen.

```
PROGRAM ZUWEISUNGEN(OUTPUT);
  VAR A,B,R: REAL;
BEGIN
  A:= 0; B:= 1.023;
  B:= B+1;
  R:= (23.7 + B * A) / (A - B * B * B)
END.
```

Da eine Zuweisung den *alten* Wert einer Variablen überschreibt, benötigt man zum Vertauschen der Werte der Variablen A und B eine Hilfsvariable H:

```
H:= A; A:= B; B:= H
```

Die Zuweisungen A:=B; B:=A alleine haben nämlich zur Folge, daß zunächst die Variable A den Wert der Variablen B erhält, und anschließend dieser Wert wieder der Variablen B zugewiesen wird.

```
(23.7 + B * A) / (A - B * B * B)
A+1
A
0
```

Syntaxregeln für Ausdrücke

sind jeweils **Ausdrücke**. Im allgemeinen enthalten Ausdrücke mehrere Operanden (z.B. Variablen, Konstanten) und Operatoren (wie +, –, OR, =). Die Struktur eines Ausdruckes in Pascal kann recht komplex werden und wird formal durch das Syntaxdiagramm AUSDRUCK im Anhang A beschrieben. Ein Ausdruck wird dort schrittweise in *einfache Ausdrücke*, *Terme*, *Faktoren* und schließlich Konstanten und Variablen zerlegt. Dabei drücken die hierarchischen Bildungsregeln für Ausdrücke gleichzeitig die Bindungsregeln der Operatoren aus. Die Regeln für Ausdrücke ähneln den Regeln in der Mathematik oder in BASIC und Fortran. Daher genügt es, auf die Besonderheiten von Pascal hinzuweisen:

1. Operatoren dürfen nicht direkt aufeinanderfolgen. Man schreibt also A + (–B) und nicht A + –B.
2. Die Multiplikationsoperatoren *, /, DIV, MOD und AND binden stärker als die Additionsoperatoren +, – und OR. In Pascal gilt also auch »Punkt- vor Strichrechnung«. Der Operator NOT bindet stärker als die Multiplikationsoperatoren und wirkt daher wie ein Vorzeichen.
   Die Additionsoperatoren binden stärker als die Vergleichsoperatoren =, <>, >=, <=, >, <, IN.
   ```
   A / 3 + Z     entspricht
   (A / 3) + Z
   NOT (X>7) OR (Y<3) AND (X=Y) entspricht
   (NOT (X>7)) OR ((Y<3) AND (X=Y))
   ```
3. Es gibt kein Operationssymbol zum Potenzieren. Der Pfeil »↑« hat eine völlig andere Bedeutung.

4. Eine Folge von Operatoren gleicher Priorität wird von links nach rechts ausgewertet:
   `A * B * C  entspricht  (A * B) * C`
   `A - B - C  entspricht  (A - B) - C`
5. Im Zweifelsfall sollte man die Priorität mit Klammern unterstreichen:
   `(A * B) - (C * D)  statt  A * B - C * D`
6. Wie in BASIC stehen auch arithmetische Standardfunktionen zur Verfügung, z.B. SIN, COS, SQRT. Im Augenblick können Sie diese Funktionen *naiv* verwenden. Alle arithmetischen Funktionen sind in der Dokumentation in Abschnitt 4.4.4.6 mit ihrer Bedeutung aufgelistet.

Semantische Regeln können in der Syntax nicht erfaßt werden

Die Operatoren werden später noch detaillierter behandelt. Dort finden Sie auch weitere Beispiele und Hinweise über die zulässigen Typen für die einzelnen Operationen. Schließlich ist es einsichtig, daß es z.B. sinnlos ist, zwei Zeichen zu addieren:

```
'A' + 'B' * 'C'
```

Solche *semantischen* Regeln lassen sich natürlich nicht mit den Syntaxdiagrammen erfassen.

2.5 Einfache Ein- und Ausgabe am Bildschirm

Ein- und Ausgabe im Vergleich zu BASIC

Zwar können Sie jetzt bereits korrekte Anweisungsfolgen bilden, in denen Sie auch Zuweisungen mit deklarierten Variablen durchführen können, jedoch fehlen Ihnen noch Möglichkeiten zur Ausgabe der Ergebnisse oder zum Einlesen variabler Werte. In diesem Abschnitt werden deshalb die Gegenstücke zu den Befehlen INPUT und PRINT in BASIC vorgestellt.
Pascal unterscheidet sich von BASIC durch ein standardmäßig vorhandenes Konzept zur Formatierung der Ausgabe und etwas vielseitigere Eingabemöglichkeiten, die eine Zeilenstruktur der Benutzereingaben berücksichtigen können.

2.5.1 WRITE

In Ihrem ersten Programm (Bild 6) trat bereits die Anweisung WRITELN auf. Syntaktisch gesehen ist WRITELN ein vordefinierter Name, dem in Klammern **Parameter**, das sind Ausdrücke getrennt durch Kommata, folgen können.
Die einfachste Form einer Ausgabeanweisung hat folgende Struktur:

```
WRITE(Ausdruck)
```

WRITE zeigt Ergebnisse an

Es genügt also, den auszugebenden Wert in Klammern hinter dem vordefinierten Namen WRITE anzugeben. Dadurch wird das Ergebnis des Ausdruckes berechnet und am Bildschirm ausgegeben:

```
PROGRAM WRITEBEISPIEL(OUTPUT);
  VAR ZWISCHENERGEBNIS: REAL;
      GANZEZAHL        : INTEGER;
BEGIN
  WRITE('Dies ist ein Beispiel ');
  WRITE('für Bildschirmausgaben:');
  WRITE(20 * 3);
  WRITE(1 / 3);
  ZWISCHENERGEBNIS:= SIN(0.5) + COS(0.5);
  GANZEZAHL:= 3;
  WRITE(ZWISCHENERGEBNIS);
  WRITE(GANZEZAHL);
  WRITE(300 > 15);
  WRITE(1, 2, 3, 4, 5, GANZEZAHL-4);
END.
```

Wenn Sie dieses Programm ausführen lassen, werden Sie feststellen, daß aufeinanderfolgende WRITE-Anweisungen die Ausgaben ohne Zwischenraum hintereinanderstellen. Die Ausgabe hängt vom **Typ** des Ausdruckes ab:

Regeln für die Ausgabe mit WRITE

1. Ist der Ausdruck ein String (z.B. eine Stringkonstante in Hochkommata) oder ein einzelnes Zeichen, so wird der String ab der momentanen Cursorposition ausgegeben. Der Cursor steht anschließend direkt hinter dem String.
2. Der Parameter ist ein arithmetischer Ausdruck, der also eine Zahl als Ergebnis liefert. Dann wird das Ergebnis der Berechnung des Ausdruckes am Bildschirm ausgegeben. Vor positiven Zahlen steht ein Leerzeichen und kein Pluszeichen (»+«). Besitzt das Ergebnis Nachkommastellen (ist es also keine ganze, sondern eine reelle Zahl), so wird die Exponentialdarstellung zur Anzeige verwendet.

   ```
   WRITE(3)          3
   WRITE(3.1)        3.10000000E+00
   WRITE(3.0)        3.00000000E+00
   ```

3. Liefert der Ausdruck einen Wahrheitswert, so wird dieser Wert mit den Worten TRUE (engl. für *wahr*) oder FALSE (für *falsch*) angezeigt. Nähere Einzelheiten zur Bedeutung und Anwendung von Wahrheitswerten werden in Abschnitt 2.6.5 genannt.

   ```
   WRITE(300 > 15)          TRUE
   WRITE(SIN(3)=COS(3))    FALSE
   ```

Zusammenfassend läßt sich also sagen, daß Sie die Ergebnisse jedes beliebigen Ausdruckes oder die Inhalte beliebiger Variablen jederzeit mit der WRITE-Anweisung darstellen können. Nicht nur für den Anfänger ist das Einfügen von WRITE-Anweisungen eine willkommene Möglichkeit, den Programmablauf am Bildschirm zu verfolgen.

Formatierung mit der Feldgröße

Speziell bei der Darstellung von Zahlen möchte man die Ausgabe oft formatiert vornehmen. Generell erfolgt eine Formatierung in Pascal durch die Angabe einer **Feldgröße**. Die Philosophie besteht darin, dem auszudruckenden Wert eine feste Anzahl an Stellen einzuräumen, innerhalb derer das Ergebnis rechtsbündig gedruckt wird. Betrachten Sie als Beispiel die Ausgabe der folgenden Tabelle:

```
    Name ¦ Vorname ¦ arbeitslos ¦ Kinder ¦ Gehalt
--------------------------------------------
   Meier ¦ Otto    ¦ FALSE      ¦ 0      ¦ 3000.00
........ ¦ ....... ¦ ......... ¦ ...... ¦ ........
```

Für die Spalten Name, Vorname, arbeitslos, Kinder und Gehalt sollen dabei die durch Punkte markierten Stellen (8, 7, 9, 6 und 8 Zeichen) bedruckt werden, während vor dem senkrechten Strich jeweils ein Leerzeichen steht.
In Pascal würde man die obige Zeile durch die Angabe einer Feldgröße nach einem Doppelpunkt hinter den zu druckenden Werten formatieren, wobei eine Person mit einem Gehalt von 0.0 als arbeitslos bezeichnet wird:

```
PROGRAM TABELLE(OUTPUT);
VAR KINDER: INTEGER;
  GEHALT: REAL;
BEGIN
  WRITELN('      Name ¦ Vorname ¦ arbeitslos ¦ Kinder ¦ Gehalt');
  WRITELN('---------------------------------------------');
  KINDER:=0; GEHALT:= 3000.0;
  WRITE('Meier'       : 8);      WRITE(' ¦ ');
  WRITE('Otto'        : 7);      WRITE(' ¦ ');
  WRITE(Gehalt = 0.0  : 9);      WRITE(' ¦ ');
  WRITE(Kinder        : 6);      WRITE(' ¦ ');
  WRITE(Gehalt        : 8 : 2);  WRITE(' ¦ ');
  WRITELN;
END.
```

Formal läßt sich die Wirkung der Angabe einer Feldgröße folgendermaßen beschreiben: Die Feldgröße definiert eine Mindestanzahl an Zeichen, die bei der Ausgabe gedruckt wird. Ist die Darstellung des auszugebenden Wertes kleiner als die Feldgröße, so wird der Wert **rechtsbündig** in einem Feld der genannten Größe ausgegeben. Die Feldgröße wird ignoriert, falls die Ausgabe zu lang ist.

Variable Feldgröße

Interessanterweise kann die Feldgröße auch durch einen Ausdruck angegeben werden, der ein ganzzahliges Ergebnis liefert. Damit ist also eine variable Feldgröße möglich.
Beispiel (die Striche stellen Leerzeichen dar, die zum Erreichen der Feldgröße eingefügt werden):

```
WRITE (3*4 : 5);        ---12            (rechtsbündig)
WRITE('***': 5);        --***            (rechtsbündig)
WRITE(3.0  : 5);        3.00000000E+00   (Feld war zu klein)
WRITE('xxx': 0);        xxx              (Feld war zu klein)
WRITE(12   : 3+2);      ---12            (3+2 = 5)
```

In den Beispielprogrammen WRITEBEISPIEL und TABELLE wurden bereits weitergehende Möglichkeiten der Ausgabe in Pascal verwendet:

Zusammengesetzte WRITE-Befehle

Beliebig viele direkt aufeinanderfolgende WRITE-Befehle können nämlich immer zu einem einzigen zusammengefaßt werden, wobei man die Parameter durch Kommata trennt. Die Werteliste hinter der WRITE-Anweisung kann sich natürlich auch über mehrere Quelltextzeilen erstrecken:

```
WRITE('Meier'      : 8,     ' ¦ ',
      'Otto'       : 7,     ' ¦ ',
      Gehalt = 0.0 : 9,     ' ¦ ',
      Kinder       : 6,     ' ¦ ',
      Gehalt       : 8 : 2, ' ¦ ');
```

Formatierung reeller Zahlen

Die letzte Ausgabe dieser Anweisung besitzt zwei Formatierungsparameter, die durch Doppelpunkte getrennt sind. Dadurch können reelle Zahlen auch im Festpunktformat dargestellt werden. Der erste Formatierungsparameter bestimmt wiederum die Feldgröße, während der zweite Parameter die Anzahl der Nachkommastellen definiert. Deshalb wird im obigen Beispiel das Gehalt rechtsbündig in einem Feld aus 8 Zeichen ausgegeben, wobei 2 Nachkommastellen gedruckt werden.

WRITELN definiert Zeilenwechsel

Im Programmbeispiel TABELLE wurden insgesamt drei Bildschirmzeilen ausgegeben. Um die Ausgabe am Anfang der Folgezeile fortzusetzen, existiert die Anweisung WRITELN (*write line*). So werden mit der Anweisungsfolge

```
WRITELN; WRITELN; WRITELN
```

drei Leerzeilen gedruckt. Für WRITELN sind die gleichen Parameter wie bei der Anweisung WRITE zulässig. Dabei entspricht die Ausgabe

```
WRITELN(Ausdruck, Ausdruck, ...)
```

folgender Anweisungsfolge:

```
WRITE(Ausdruck, Ausdruck, ...); WRITELN
```

Es wird also zusätzlich am Ende der Ausgabe ein Zeilenvorschub ausgeführt (siehe auch Beispielprogramm TABELLE).

2.5.2 READ

Eingabe unformatiert mit Zeilenwechsel

Die Eingabe ist in Pascal völlig symmetrisch zur Ausgabe organisiert. Jedoch entfallen die Formatierungsparameter, so daß es nicht wie in Fortran oder COBOL möglich ist, eine Eingabezeile in Felder fester Länge einzuteilen.

Haben Sie im Vereinbarungsteil, wie im letzten Kapitel beschrieben, Variablen deklariert, so können Sie auch Werte vom Bildschirm einlesen. Dabei erfolgt die Eingabe zeilenweise.

Eingaben zur Nullstellenberechnung

```
PROGRAM PQFORMEL(INPUT, OUTPUT);
  VAR P,Q,W,A: REAL;
BEGIN
  WRITE('P Q =');
  READ(P);  READ(Q);
  READLN;
  W:= SQRT(P * P / 4 - Q);
  A:= - P / 2;
  WRITELN('X1=', A+W);
  WRITELN('X2=', A-W);
END.
```

Mit diesem Programm können Sie Nullstellen der gemischtquadratischen Funktion

```
0 = X * X + P * X + Q
```

bestimmen. Der Benutzer gibt die Werte P und Q vor, zu denen das Programm die Nullstellen X1 und X2 berechnen soll. Falls Sie sich noch an Ihren Mathematikunterricht erinnern, werden Sie auch wissen, daß nicht immer eine (reelle) Lösung existiert. Dieser Fall wird durch das Programm nicht überprüft, so daß es in diesem Fall versuchen würde, eine Quadratwurzel aus einer negativen Zahl zu berechnen.

READ belegt Variablen

Während Sie im Programm die Zuweisungen und die Ausgabeanweisung mit WRITELN bereits verstehen können, sind die Eingabeanweisungen READ und READLN Ihnen noch nicht geläufig. Eine Eingabe erfolgt mit der Anweisung READ, gefolgt von dem Namen einer deklarierten Variablen:

```
READ (Variablenname)
```

Der Benutzer wird mit blinkendem Cursor zur Eingabe aufgefordert. Er kann dann eine beliebige Folge von Zeichen eingeben. Schließt er die Eingabe mit der Return-Taste ab, so beginnt der Computer mit der Auswertung des Inhaltes der eingegebenen Zeile. Dabei bearbeitet er die Zeile sequentiell von links nach rechts.

Im Augenblick soll nur die Eingabe von Zahlen und Zeichen untersucht werden. Das Einlesen von Strings wird in Abschnitt 2.9.2 beschrieben.

1. Die Variable ist vom Typ CHAR und speichert also einzelne Zeichen. In diesem Fall wird das nächste Zeichen der Zeile gelesen und der Variablen zugewiesen.
2. Die Variable ist vom Typ INTEGER oder REAL und speichert also ganze oder reelle Zahlen. Dann werden zunächst vorlaufende Leerstellen überlesen. Nachfolgende Ziffern werden bis zum nächsten Leerzeichen gelesen.

Anschließend wird der Variablen der Wert dieser Ziffernfolge zugewiesen. Bei der Eingabe ist sowohl eine Festkommadarstellung als auch eine Fließkommadarstellung möglich.

Wird bei (1) oder (2) das Ende der Eingabezeile erreicht, so wird der Benutzer erneut zur Eingabe einer weiteren Zeile aufgefordert. Geben Sie also im Programm PQFORMEL keine Ziffern ein, sondern betätigen Sie nur die Return-Taste, so erscheint in der Folgezeile wiederum ein blinkender Cursor, bis Sie insgesamt zwei Zahlen eingegeben haben.

Erweiterte READ-Befehle

Wie bei der Ausgabe können auch zwei aufeinanderfolgende READ-Anweisungen zu einer einzigen zusammengefaßt werden. Die beiden folgenden Zeilen liefern also dieselbe Eingabe:

```
READ(P); READ(Q)
READ(P, Q);
```

Die Parameterliste einer READ-Anweisung kann demnach beliebig lang sein. Die Zahlen müssen vom Benutzer bei der Eingabe durch mindestens ein Leerzeichen oder einen Zeilenwechsel getrennt werden:

```
PROGRAM READVIEL(INPUT,OUTPUT);
  VAR A,B,C,D,E: INTEGER;
BEGIN
  READ(A,B,C,D,E);
  WRITELN('EINGELESEN WURDEN FOLGENDE ZAHLEN:');
 WRITE(A:4; B:4, C:4, D:4, E:4);
END.
```

Für dieses Programm sind also folgende Eingaben möglich:

```
30    33    44    77    0          (Return-Taste)
oder
30    33    44                     (Return-Taste)
77                                 (Return-Taste)
0                                  (Return-Taste)
```

Gibt man zu viele Werte ein,

```
30    33    44    77    0    1234  (Return-Taste)
```

so lesen die READ-Anweisungen bis zum Leerzeichen hinter der Zahl 0. Eine nachfolgende READ-Anweisung würde die Zahl 1234 lesen. Möchte man jedoch den Rest einer Eingabezeile ignorieren, so verwendet man den Befehl READLN (*read line*). Ohne weitere Parameter überliest er alle Zeichen bis zum Ende der laufenden Eingabezeile. So wurde im Beispielprogramm PQFORMEL folgende Anweisungsfolge verwendet:

READLN prüft auf Zeilenwechsel

```
READ(P); READ(Q); READLN
```

Gibt der Benutzer nämlich zu viele Werte (also mehr als 2) ein, so werden die überflüssigen Eingaben durch READLN überlesen und die nächste Eingabe wird am Anfang der Folgezeile erwartet. Wie bei WRITELN läßt sich eine

solche Folge von READ- und READLN-Anweisungen zu einem Befehl zusammenfassen:

```
READLN(P, Q);
```

Aufgaben

1. Damit Sie sich einen praktischen Überblick über die vielen verschiedenen Möglichkeiten zur Ein- und Ausgabe verschaffen können, sollten Sie die Beispiele aus dem Text am Computer ausprobieren. Dabei dürfen Sie jedoch die Deklaration der Variablen nicht vergessen. Wählen Sie verschiedene Ausdrücke und Variablen, und verändern Sie die Spaltenbreite im Programm TABELLE.
2. Ändern Sie das Programm PQFORMEL so, daß die Werte für P und Q vom Benutzer in zwei aufeinanderfolgenden Zeilen eingelesen werden, und die Ausgabe folgendermaßen formatiert wird:

   ```
   P =  ........
   Q =  ........
   X1 = 999999999.99
   X2 = 999999999.99
   ```
3. Schreiben Sie ein Programm, das den Text **Überschrift** in einem Feld aus N Zeichen **zentriert** druckt. Das Feld soll durch zwei Sterne links und rechts begrenzt werden. Dabei soll der Benutzer die Zahl N über die Tastatur eingeben können:

   ```
   N= 30
   *..........ÜBERSCHRIFT.........*
   N= 20
   *.....ÜBERSCHRIFT....*
   ```

   Hinweis: Die Idee besteht darin, zunächst zu berechnen, wie viele Leerzeichen links und rechts eingefügt werden müssen. Die Anzahl der Leerstellen speichern Sie in zwei Variablen (LINKS und RECHTS) und benutzen anschließend folgende Anweisungen:

   ```
   PROGRAM CENTER(INPUT,OUTPUT);
     VAR N, LINKS, RECHTS: INTEGER;
   BEGIN
     (* Hier müssen Sie N einlesen *)
     (* Hier berechnen Sie LINKS und RECHTS *)
     WRITELN('*',
            ' ': LINKS,
            'ÜBERSCHRIFT',
            ' ': RECHTS,
           '*');
   END.
   ```

4. Es werden vom Benutzer folgende drei Zeilen eingegeben:

1	2	3	4
5	6	7	8
9	10	11	12

   Programmieren Sie drei Anweisungsfolgen, die folgende Werte lesen:
   1. Die Zahlen 1, 2, 3, 4 und 9
   2. Die Zahlen 1, 5 und 9
   3. Die Zahlen 4 und 9

   Sie müssen also geeignete READ- und READLN-Anweisungen verwenden, die die übrigen Werte überlesen.
5. Ist Ihnen die genaue Wirkung der Feldgröße für reelle Zahlen noch unklar, so sollten Sie ein wenig mit folgendem Programm experimentieren (detaillierte Regeln für die Ausgabe finden sich übrigens nochmals in der Dokumentation in Abschnitt 4.4.4.5):

```
PROGRAM WRITEREAL(INPUT,OUTPUT);
 VAR R: REAL;
    FELDGROESSE: INTEGER;
    NACHKOMMA  : INTEGER;
BEGIN
 READLN(R, FELDGROESSE, NACHKOMMA);
 WRITELN('!', R: FELDGROESSE : NACHKOMMA, '!');
END.
```

6. Experimentieren Sie mit Ausdrücken, Anweisungsfolgen und den Ein- und Ausgabebefehlen! Schreiben Sie eigene kleine Programme, um ein Gefühl für Ausdrücke in Pascal zu bekommen!
   Sollten Sie keine eigenen Ideen haben, können Sie sich an der folgenden kleinen Liste orientieren: Berechnung von Zinsen und Zinseszinsen, Berechnung der Fläche von Kreisen und Ellipsen, des Volumens von Kugeln und Kegeln. Berechnung des Logarithmus zur Basis 10. Vergleichen Sie
   `SIN(X) / COS(X)  mit TAN(X)`
   Wie groß sind die Rechenfehler des Computers bei der Gleichung
   `SQR(SIN(X)) + SQR(COS(X)) = 1 ?`
   Wie berechnet man die Umkehrfunktion von SIN(X)? Eine Lösung findet sich übrigens im BASIC-Handbuch.

2.6 Elementare Datentypen

In den vorangegangenen Abschnitten trat bereits der Begriff **Typ** auf: Zahlen wurden unterschieden in ganze Zahlen und reelle Zahlen, Variablen wurden bei der Deklaration an einen Typ gebunden, und Ausdrücke besaßen einen Typ. Falls Sie bereits in BASIC programmiert haben, wissen Sie, daß es dort nur

Typen in Pascal und BASIC

zwei Typen gibt: reelle Zahlen und Strings. Einige BASIC-Dialekte erlauben auch eine Unterscheidung zwischen ganzen und reellen Zahlen. Die Typen von Variablen gehen in BASIC aus dem Suffix des Variablennamens hervor (»$« oder »%«).
In Pascal gibt es auch Objekte ganz anderer Typen. Später werden Sie sogar lernen, wie man eigene Typen in Pascal definiert (Abschnitt 2.12). Dieser Abschnitt beschäftigt sich mit den elementaren **Standardtypen**. Diese Typen besitzen in Pascal vordefinierte Namen:

Typname	Wertebereich	Beispielwerte
INTEGER	Ganze Zahlen	0 –7 500 3256
REAL	Reelle Zahlen	3.25 0.0 –5E–13
CHAR	Einzelzeichen	'A' 'O' '&'
STRING	Zeichenketten	'Alpha' '12345678'
BOOLEAN	Wahrheitswerte	TRUE FALSE

Nachfolgend werden die Eigenschaften jedes Typs im einzelnen besprochen und die Operationen mit Objekten dieser Typen vorgestellt.

2.6.1 Der Typ INTEGER

Werte vom Typ INTEGER sind ganze Zahlen, also positive und negative Zahlen ohne Nachkommastellen. Die wichtigsten Operationen, die auf ganze Zahlen anwendbar sind, liefern als Ergebnis wieder eine ganze Zahl:

Operationen mit ganzen Zahlen

+	Identität oder Addition	+7	3+4
–	Subtraktion oder Vorzeichenwechsel	–3	1–4
*	Multiplikation	12*43	
DIV	ganzzahlige Division	44 DIV 13	
MOD	Modulo-Bildung (Divisionsrest)	44 MOD 13	

Um die Wirkungsweise der letzten beiden Operationen zu verdeutlichen, folgen noch einige Zahlenbeispiele:

10	DIV	3	=	3	10	MOD	3	=	1
(–10)	DIV	3	=	–3	(–10)	MOD	3	=	–1
10	DIV	(–3)	=	–3	10	MOD	(–3)	=	1
(–10)	DIV	(–3)	=	3	(–10)	MOD	(–3)	=	–1

X DIV Y ist also der ganzzahlige Anteil des Quotienten X/Y. X MOD Y ist gleich dem Rest der Division X DIV Y. Formal hängen DIV und MOD folgendermaßen zusammen:
X = (X DIV Y) * Y + (X MOD Y)

Weiterhin existiert die arithmetische Funktion ABS(n), die den Absolutwert (Betrag) der Zahl n liefert:

ABS(3) = 3 ABS(–3) = 3 ABS(0) = 0

Jeder Computer kann nur Zahlen einer endlichen Größe darstellen. In *Pascal2.0* sind als ganze Zahlen nur Werte mit

–MAXINT–1 <= N <= MAXINT

Der Wertebereich für ganze Zahlen

zugelassen. MAXINT ist der vordefinierte Name für eine Konstante mit dem Wert

MAXINT = 32767

Tritt bei einer der obigen Operationen mit ganzen Zahlen eine Bereichsüberschreitung auf, so meldet dies das Pascal-Laufzeitsystem und unterbricht das laufende Programm:

```
PROGRAM UEBERLAUF(INPUT,OUTPUT);
BEGIN
  WRITELN(1000*1000 DIV 1000);
END.
```

Obwohl das mathematische Endergebnis eine gültige INTEGER-Zahl ist, wird während der Berechnung der darstellbare Zahlenbereich verlassen, was der Computer mit folgender Meldung quittiert:

```
INTEGER OVERFLOW
ERROR AT 7418 IN UEBERLAUF
```

2.6.2 Der Typ REAL

Operationen mit reellen Zahlen

Werte des Typs REAL sind reelle Zahlen. Die arithmetischen Operationen (+, –, *, /) liefern angewandt auf reelle Zahlen ein Ergebnis vom Typ REAL. Der Schrägstrich »/« liefert also das normale Ergebnis einer Division:

1.5 / 1.2 = 1.25

Andererseits dürfen die Operationen MOD und DIV nicht auf Werte vom Typ REAL angewendet werden. Weiterhin sind alle üblichen arithmetischen Funktionen in Pascal definiert. Sie liefern jeweils ein Ergebnis vom Typ REAL:

Name in Pascal	Operation	Name in BASIC
ABS	Absolutwert	ABS
SQRT	Quadratwurzel	SQR
EXP	Exponentialfunktion	EXP
LN	natürlicher Logarithmus	LOG
SIN	sinus	SIN
COS	cosinus	COS
ARCTAN	arcus tangens	ATN
SQR	Quadrat	–

Winkel in Bogenmaß und Grad

Die trigonometrischen Funktionen sind für Winkel in Bogenmaß definiert. Wollen Sie mit Winkeln in Grad arbeiten, so müssen Sie zunächst eine Umrechnung vornehmen. Es gilt:

$180° = \pi$ (rad)

Die simple Dreisatzrechnung zur Umrechnung sieht in Pascal folgendermaßen aus:

```
 PROGRAM WINKEL (INPUT, OUTPUT);
 VAR X, T     : REAL;
     FAKTOR1 : REAL;
BEGIN
 WRITE('Winkel (in Grad):'); READLN(X);
 FAKTOR1:= 1.74532925E-2;          (* das ist PI/180  *)
 WRITELN('SIN(',X :5 :2,') =',
         SIN(X * FAKTOR1));
 WRITELN;
 WRITE('Tangens:'); READLN(T);
 WRITE('Der Winkel ', ARCTAN(T) / FAKTOR1 :5 :2);
 WRITELN(' besitzt den Tangens', T);
END.
```

Erweiterungen in Pascal 2.0

Zusätzlich zu den in der Tabelle aufgeführten Funktionen existieren in *Pascal 2.0* zwei weitere Funktionen, die nicht im *report* vorgesehen sind. Beide Funktionen liefern ein Ergebnis vom Typ REAL:

POWER(x,y) Diese Funktion berechnet x hoch y für beliebige reelle Zahlen x und y. Die Berechnung ist recht aufwendig und erfolgt intern nach der Formel:
POWER (x,y) = EXP(y * LN(x))
Daher ist für kleine ganzzahlige y eine Berechnung durch mehrmalige Multiplikation schneller. Also
X*X*X statt POWER (X,3)

TAN(x) Diese Funktion berechnet den Tangens des Winkels x in Bogenmaß.

Jede Implementierung setzt auch eine Grenze für den Zahlenbereich, in dem reelle Zahlen dargestellt werden können. Um die Ergebnisse von Operationen mit reellen Zahlen zu verstehen, muß man die interne Darstellung von Werten des Typs REAL kennen.

Festkommazahlen sind unflexibel

Würde man sich z.B. darauf einigen, daß von jeder reellen Zahl 8 Stellen vor dem Komma und 4 Nachkommastellen gespeichert werden, so bestünde der darstellbare Zahlenbereich aus den Zahlen zwischen

-999999999.9999
+999999999.9999

Dabei müßten alle Ziffern, die mehr als vier Stellen hinter dem Komma stehen, ignoriert werden. Offensichtlich bietet eine solche willkürliche Beschränkung auf Vor- und Nachkommastellen für viele Anwendungen keine befriedigende Lösung.
Wie speichert man also so verschiedene Zahlen wie
A = 9876543219876543210
B = 123000000000
C = 0.0000000000234
am günstigsten? Die Idee für eine kompakte Speicherung der Zahlen besteht darin, sich zunächst die Größenordnung der Zahl zu merken: A besitzt 19 Stellen vor dem Komma, B hat 13 Vorkommastellen, während C an der 12. Stelle hinter dem Komma beginnt. Außerdem werden von jeder Zahl die ersten sogenannten *signifikanten* Stellen gespeichert.

Fließkommazahlen beim C 128

Beim C 128 kann eine Zahl maximal 38 Stellen vor oder nach dem Komma beginnen. Von jeder Zahl werden jedoch maximal 9 Ziffern gespeichert. An der neunten Stelle wird gerundet. Intern wird jede reelle Zahl durch zwei Werte (*Mantisse* und *Exponent*) dargestellt:

Mantisse	Exponent
A =	0.987654322E+19
B =	0.123000000E+12
C =	0.234000000E–10

Die Länge der Mantisse bestimmt also die Genauigkeit einer Zahlendarstellung, während die Größe des Exponenten die maximale Größe der darstellbaren Zahlen begrenzt. Für den C 128 gelten folgende rechnerspezifischen Regeln:
- Zahlen größer als +/– 10 hoch 38 sind nicht darstellbar
- Zahlen kleiner als +/– 10 hoch –38 werden als 0 dargestellt
- Bei allen Zahlen wird nach der neunten Stelle gerundet

Probleme mit Rundungsfehlern bei Fließkommazahlen

Diese Grenzen werden in der Praxis mit einem Heimcomputer nie überschritten, außer man wollte z.B. DM-Beträge über 9 Millionen auf den Pfennig genau speichern. Problematisch ist aber nicht die Speicherung der Zahlen, sondern das Rechnen mit REAL-Zahlen. Da auch alle Zwischenergebnisse *nur* auf neun Stellen genau sind, liefern scheinbar harmlose Berechnungen falsche Ergebnisse:
Mit den obigen Zahlen ist zum Beispiel C + B – B nicht gleich C, da der Computer folgendermaßen rechnet:
C + B – B = (C + B) – B = 1.23E+11 – 1.23E+11 = 0
Bei Berechnungen mit REAL-Zahlen sollte man sich solcher *Ungenauigkeiten* immer bewußt sein und zuerst Werte der gleichen Größenordnung verknüpfen.

Besonders kritisch sind Vergleiche zwischen reellen Zahlen. Man sollte deshalb nie auf Gleichheit zweier reeller Zahlen prüfen:

```
IF A = B THEN ....      (falsch!)
```

Besser prüft man, ob die Differenz zwischen A und B innerhalb einer durch die Rundungsfehler gegebenen Toleranzzone liegt:

```
IF ABS(A-B) <= 1E-7 THEN ...     (besser!)
```

2.6.3 Gegenüberstellung REAL und INTEGER

Solange alle Zwischenergebnisse in dem durch die Konstante MAXINT angegebenen Bereich liegen, sind Operationen mit Werten vom Typ INTEGER im mathematischen Sinn exakt. Außerdem werden ganze Zahlen kompakter als reelle Zahlen gespeichert. In *Pascal 2.0* benötigt eine ganze Zahl nur zwei Speicherstellen gegenüber fünf Speicherstellen für reelle Zahlen. Nicht zuletzt sind INTEGER-Operationen wesentlich effizienter (kürzerer Code, höhere Ausführungsgeschwindigkeit) als solche mit reellen Zahlen.

Geschwindigkeitsvergleich REAL- und INTEGER-Arithmetik

```
PROGRAM SPEED (INPUT,OUTPUT);
(* Dieses Programm zeigt die Geschwindigkeitsunterschiede *)
(* zwischen Operationen mit reellen und ganzen Zahlen     *)
  VAR ZAEHLER: INTEGER;
      I      : INTEGER;
      R      : REAL;
BEGIN
  WRITELN('Schleife (REAL) gestartet' #7);
  R:=0;
  FOR ZAEHLER:= 0 TO 1000 DO
    R:= (ZAEHLER - R) / 2;
  WRITELN('Schleife (INTEGER) gestartet' #7);
  I:= 0;
  FOR ZAEHLER:= 0 TO 1000 DO
    I:= (ZAEHLER - I) DIV 2;
  WRITELN('Schleife beendet' #7);
  WRITELN('Ergebnisse:', R, I);
END.
```

Andererseits können große Zahlen nur mit Werten vom Typ REAL dargestellt werden. Auch alle höheren Funktionen liefern Werte vom Typ REAL als Ergebnisse.

Programmsteuerung nur mit INTEGER-Zahlen

Zusammenfassend läßt sich sagen, daß reelle Zahlen nur in mathematischen Anwendungen (Nullstellenbestimmungen, statistischen Auswertungen etc.) und in kaufmännischen Programmen zur Darstellung großer Zahlen verwendet werden. Typische Anwendungen für ganze Zahlen sind hingegen Steuerungsaufgaben im Programm wie Zähler, Indizes und Laufvariablen. Nicht umsonst

sind mindestens zwei Drittel aller Variablen in typischen Pascal-Programmen vom Typ INTEGER.

Automatische Typanpassung in gemischten Ausdrücken

Während Sie schon einige Fälle kennengelernt haben, in denen **keine** REAL-Werte zulässig sind, nämlich als Feldgröße bei WRITE oder als Operand bei den Operatoren DIV und MOD, können umgekehrt **überall** dort, wo reelle Zahlen erlaubt sind, auch ganze Zahlen stehen. Der Compiler erzeugt an diesen Stellen Code zur Umwandlung in die Darstellung mit Mantisse und Exponent.

```
SIN(3)     3/4     SQRT(2)
```

Eine solche Umwandlung von INTEGER- in REAL-Zahlen erfolgt auch bei den folgenden **gemischten** Ausdrücken, bei denen eine INTEGER-Zahl mit einer reellen Zahl verknüpft wird:

```
3 + 4.0     1 + 2 + 4 * 20.7
```

Das Ergebnis eines gemischten Ausdruckes ist immer vom Typ REAL.

Explizite Typanpassung für reelle Zahlen

Die umgekehrte Umwandlung von reellen Zahlen in ganze Zahlen muß explizit programmiert werden, damit festgelegt werden kann, wie die Nachkommastellen behandelt werden. In Pascal sind dazu die Funktionen ROUND und TRUNC vorhanden. ROUND(x) rundet das reelle Argument, während TRUNC nur die Nachkommastellen abschneidet. Pascal 2.0 bietet zusätzlich die Funktion INT, die das reelle Argument zur nächstkleineren ganzen Zahl abrundet. Beispiele zeigen am besten die unterschiedlichen Ergebnisse:

```
ROUND( 3.2) =  3   TRUNC( 3.2) =  3   INT( 3.2) =  3
ROUND( 3.7) =  4   TRUNC( 3.7) =  3   INT( 3.7) =  3
ROUND(-3.2) = -3   TRUNC(-3.2) = -3   INT(-3.2) = -4
ROUND(-3.7) = -4   TRUNC(-3.7) = -3   INT(-3.7) = -4
```

2.6.4 Der Typ CHAR

Nur ein geringer Teil aller Programme arbeitet ausschließlich mit Zahlen. Eine Klasse von Objekten, die vor allem bei der Kommunikation des Rechners mit seiner Umwelt eine große Rolle spielen, sind **Zeichen**.

Einzelzeichen als Datentyp

Werte des Typs CHAR (*character*) sind **einzelne** Zeichen. Jedes Zeichen besitzt eine Ordnungsnummer (Codenummer). Der Zusammenhang zwischen Zeichen und Ordnungsnummer ist leider vom jeweiligen Rechner abhängig. Speziell auf Commodore-Rechnern gibt es 256 verschiedene Zeichen. Eine Variable vom Typ CHAR kann also genau eines dieser 256 Zeichen speichern. Nur 160 dieser Zeichen sind auch am Bildschirm darstellbar. Die restlichen Zeichen (Kontrollzeichen) erfüllen spezielle Aufgaben bei den einzelnen Geräten.

So besitzen die Cursortasten bei Tastatureingaben ein eigenes Zeichen, mit einigen Zeichen läßt sich bei der Bildschirmausgabe die Zeichenfarbe wählen, und wieder andere Zeichen wählen den Schrifttyp am Drucker.

Konstanten vom Typ CHAR sind Stringkonstanten (siehe Definition in Abschnitt 2.1) der Länge 1:

'.', '*', 'A', #147 und 'x'

Vergleiche zwischen Einzelstücken

Vergleiche sind die einzigen Operationen, die zwischen Zeichen definiert sind. Das Ergebnis eines Vergleichs zweier Zeichen ist durch ihre Ordnungszahl festgelegt. Eine Liste aller Ordnungszahlen findet sich im Anhang A des BASIC-Handbuchs. Auf Commodore-Rechnern gilt also:

'A' < 'Z'
'0' < '9'
'!' < ')'
'b' < 'B'

Der letzte Vergleich ist besonders verwirrend, da auf einem Commodore-Computer Kleinbuchstaben nur durch Umschaltung der Bildschirmdarstellung erreicht werden können. Dann besitzen die mit der Shift-Taste eingegebenen Großbuchstaben eine höhere Codenummer als die normal eingegebenen Kleinbuchstaben.

Typumwandlungen zwischen CHAR und INTEGER

Innerhalb eines Pascal-Programmes können Sie die Ordnungsnummer jedes Zeichens mit der Standardfunktion ORD erhalten:

ORD('A')	=	65	ORD('Z')	=	90
ORD('0')	=	48	ORD('9')	=	57
ORD('!')	=	33	ORD(')')	=	41
ORD('b')	=	66	ORD('B')	=	194

Die Umkehrung der Funktion ORD ist die Funktion CHR: Sie liefert zu einem Argument vom Typ INTEGER das Zeichen mit der angegebenen Ordnungsnummer:

CHR(65) = 'A' CHR(57) = '9'

Diese Umwandlung zwischen Zeichen und Ordnungsnummer ist, wie bereits erwähnt wurde, vom zugrundeliegenden Zeichensatz abhängig. Jedoch können Sie davon ausgehen, daß auf jedem Rechner die Zahlen und Buchstaben zusammenhängend aufsteigend geordnet sind, so daß '0' < '9' und 'A' < 'Z' ist. Eine recht häufige Anwendung dieser Umwandlungsroutinen ist der selektive Zugriff auf einzelne Ziffern einer Zeichenfolge. Das folgende Beispielprogramm berechnet die Quersumme einer zweistelligen Zahl, die der Benutzer eingibt:

Arithmetik mit Einzelzeichen

```
PROGRAM QUERSUMME (INPUT,OUTPUT);
  VAR ZIFFER1,ZIFFER2: CHAR;
        WERT1, WERT2: INTEGER;
BEGIN
```

```
  READLN(ZIFFER1, ZIFFER2);
  WERT1:= ORD(ZIFFER1) - ORD('0');
  WERT2:= ORD(ZIFFER2) - ORD('0');
  WRITELN('Quersumme ', WERT1 + WERT2);
END.
```

Steuerzeichen

Durch die Möglichkeit von Pascal 2.0, auch Steuerzeichencodes als Zeichenkonstanten zu definieren, könnten Sie z.B. mit dem folgenden kleinen Programm die Cursortasten abfragen. Dabei werden alle Möglichkeiten zur Umwandlung zwischen den Typen INTEGER und CHAR verwendet. Die »Befehle« REPEAT und UNTIL bewirken eine Wiederholung der eingerückten Befehle, bis die Abbruchbedingung hinter UNTIL zutrifft. Es werden also so lange Eingaben von der Tastatur mit dem »Befehl« GETKEY gelesen, bis der Benutzer die Return-Taste betätigt.

```
PROGRAM TASTATURABFRAGE (INPUT,OUTPUT);
(* Die RETURN-Taste besitzt den Code 13 *)
  VAR CH: CHAR;
BEGIN
  WRITELN('Tastaturabfrage mit GETKEY (Ende mit RETURN)');
  REPEAT
   GETKEY(CH);
   IF CH = CHR(29)  THEN WRITE ('Cursor rechts');
   IF ORD(CH) = 157 THEN WRITE ('Cursor links');
   WRITE(CH);
  UNTIL CH = #13;            (* RETURN-Taste betätigt *)
 WRITELN('Programm beendet.');
END.
```

2.6.5 Der Typ BOOLEAN

Wahrheitswerte als Konstanten des Typs BOOLEAN

Zur Steuerung des Programmablaufs in Abhängigkeit von bestimmten Bedingungen sind Wahrheitswerte erforderlich. Wahrheitswerte werden in Pascal durch TRUE (wahr) und FALSE (falsch) beschrieben. Formal sind TRUE und FALSE Konstanten des Typs BOOLEAN. Viele Operationen liefern Wahrheitswerte als Ergebnis. Die Relationen zwischen Zahlen und Zeichen wurden bereits angesprochen:

(17 = 0)	FALSE	'X' < 'Y'	TRUE
(17 > 0)	TRUE	'!' < 'A'	TRUE
(0.5=5E–1)	TRUE	'X' = 'x'	FALSE

Die Funktion ODD

Ein weiteres Beispiel ist die Funktion ODD(n), die den Wert TRUE liefert, falls das Argument n ungerade ist:

ODD(3) TRUE
ODD(16) FALSE
ODD(0) FALSE

Gibt man bei der Deklaration einer Variablen den Typ BOOLEAN, so kann man ihr die Ergebnisse solcher Vergleiche zuweisen. Boolesche Variablen können natürlich ihrerseits wieder in Ausdrücken auftauchen. Sind B1 und B2 zwei logische Ausdrücke, so kann man mit den logischen Operatoren AND, OR, NOT neue Ausdrücke bilden:

Logische Operatoren

B1 AND B2 TRUE, falls B1=TRUE und B2=TRUE ist
B1 OR B2 TRUE, falls B1=TRUE oderB2=TRUE ist
NOT B1 TRUE, falls B1=FALSE ist

Wenn Sie jetzt wieder einmal die Syntaxdiagramme im Anhang A (EINFACHER AUSDRUCK und TERM) betrachten, werden Sie feststellen, daß dort diese logischen Operatoren mit den arithmetischen Operatoren (+, –, * etc.) aufgeführt sind. AND ist ein Multiplikationsoperator und bindet daher stärker als der Additionsoperator OR. NOT ist hingegen wie ein Vorzeichen definiert.

Komplexe boolesche Ausdrücke

Diese Definition unterscheidet Pascal von vielen anderen Sprachen, da hierdurch AND, OR und NOT stärker binden als die Relationen =, >, < etc. Das folgende Beispielprogramm zeigt sowohl kompliziertere boolesche Ausdrücke, als auch die Verwendung boolescher Variablen und die Ausgabe von Wahrheitswerten.

```
PROGRAM BOOL (INPUT,OUTPUT);
  VAR P,Q                  : BOOLEAN;
     ALLESFALSCH, ZUGROSS  : BOOLEAN;
     I, J                  : INTEGER;
BEGIN
  READLN(I,J);
  P:= TRUE; WRITELN(P);
  Q:= I>J; WRITELN('I>J ', Q);
  WRITELN('BEIDE UNGERADE: ', ODD(I) AND ODD(J));
  P:= (I=J) OR Q;
  ALLESFALSCH:= NOT P AND NOT Q;
  ZUGROSS:= (I>500) OR (J>500);
  WRITELN(P, ALLESFALSCH, ZUGROSS);
  IF P AND ALLESFALSCH OR (I=0) THEN
  WRITELN('Dieser Text wird nur ausgegeben, falls I=0 ist');
END.
```

Sie sehen also, daß man mit wenigen Operationen, die boolesche Variablen beinhalten, sehr undurchsichtige Bedingungen formulieren kann. Um die exakten Ausgaben für beliebige Werte von I und J vorherzusagen, müssen Sie

sicher noch einmal die Prioritätsregeln für die Operatoren AND, OR und NOT studieren.

Ungewöhnliche Prioritätsregeln

Sehr wichtig ist die Tatsache, daß Teilausdrücke, die Vergleiche enthalten, innerhalb boolescher Ausdrücke geklammert werden müssen. Man schreibt also:

```
(I=J) OR (I<>K)
```

und **nicht**

```
I=J OR I<>K
```

da der Compiler OR wie einen Additionsoperator behandelt und daher den zweiten Ausdruck implizit folgendermaßen klammern würde:

```
I = (J OR I) <> K
```

Tips für gut lesbare Programme

Das Beispielprogramm verdeutlicht auch eine Namenskonvention: Man bezeichnet boolesche Variablen meist mit Adjektivnamen, die den Wahrheitswert symbolisieren, den die Variable speichert:

```
ZUGROSS:= (I>500) OR (J>500)
```

Nur selten wird die Tatsache ausgenutzt, daß die Funktion ORD auch auf boolesche Argumente anwendbar ist. Dadurch ist auch der Typ BOOLEAN geordnet, wobei gilt:

```
ORD(FALSE) = 0     ORD(TRUE) = 1
```

und

```
FALSE < TRUE
```

Eine Bitte am Schluß: Schreiben Sie in einem booleschen Ausdruck nicht

```
P = TRUE     oder     ALLESFALSCH = FALSE
```

Dies ist zwar syntaktisch und semantisch völlig korrekt, zeugt aber von einem schlechten Stil. Man schreibt einfacher und deutlicher:

```
P            oder     NOT ALLESFALSCH
```

2.6.6 Der Typ STRING

Strings als Spracherweiterung

Der in diesem Abschnitt vorgestellte Typ gehört **nicht** zu dem im *report* definierten Standardsprachumfang. Jedoch wird er von vielen Compilern, für Mikrocomputer seien nur *Turbo-Pascal* und *UCSD-Pascal* genannt, unterstützt. Mit diesem Quasi-Standard geht auch der auf Diskette mitgelieferte Compiler Pascal 2.0 konform.

Im Gegensatz zu den übrigen in Kapitel 2.6 vorgestellten Typen ist der Typ STRING eigentlich kein elementarer Typ. Ein Wert vom Typ STRING besteht nämlich aus einer variablen Anzahl von Werten des Typs CHAR. Somit sind die folgenden Zeichenketten mit 0, 16 und 10 Zeichen Länge alle vom Typ STRING.

```
''
'---> * <---'
'LAGERREGAL'
```

Vergleiche zwischen Strings

Werte des Typs STRING können wie Einzelzeichen auch miteinander verglichen werden. Diese Vergleiche liefern ebenfalls ein Ergebnis vom Typ BOOLEAN:

'MAUS' > 'KATZE'	TRUE
'PASCAL' = 'PASCAL '	FALSE
'ANNABELLE' > 'ANNA'	TRUE

Ein String S1 ist kleiner als ein String S2, falls er in der üblichen lexikographischen Ordnung vor S2 auftritt. Natürlich ist das Vergleichsergebnis ebenfalls vom zugrundeliegenden Zeichensatz des Computers abhängig. Es ist wichtig zu beachten, daß Leerzeichen in Strings bei Vergleichen berücksichtigt werden. Daher ist 'PASCAL' < > 'PASCAL '. Um die Ergebnisse von Vergleichen auszuprobieren, können Sie das folgende kleine Programm verwenden:

```
PROGRAM STRINGORDNUNG(INPUT,OUTPUT);
  VAR S1,S2: STRING;
BEGIN
  READLN(S1);
  READLN(S2);
  IF S1>S2 THEN
    WRITELN(S1, ' > ', S2)
  ELSE
    WRITELN(S1, ' <= ', S2);
END.
```

Das Programm zeigt auch die Eingabe und Ausgabe von String-Variablen. In *Pascal 2.0* existieren zahlreiche String-Operationen, die in Kapitel 2.9.2 vorgestellt werden. Dort wird ebenfalls erklärt, wie man gezielt auf einzelne Zeichen (vom Typ CHAR) eines Strings zugreifen kann.

Aufgaben:

1. Prüfen Sie die Bereichsgrenzen für ganze und reelle Zahlen in *Pascal 2.0.* Welche Fehlermeldungen erhalten Sie? Lokalisieren Sie die Fehler im Quelltext mit der Option LOCATE ADDRESS des Compilers!
2. Wie muß man in Abschnitt 2.6.2 den Ausdruck C+B–B klammern oder umstellen, um ein korrektes Ergebnis zu erhalten?
3. In der Dokumentation ist die Funktion RANDOM(x) von *Pascal 2.0* beschrieben, die eine reelle Zufallszahl größer oder gleich Null und kleiner als Eins liefert. Schreiben Sie ein Programm, das **ganzzahlige** Zufallszahlen berechnet!

3.1 Zunächst simulieren Sie eine Münze, die die Werte 0 und 1 mit gleicher Wahrscheinlichkeit liefert.

3.2 Erweitern Sie den Wertebereich, so daß ein Würfel simuliert wird (Werte 1, 2, 3, 4, 5 und 6 gleichwahrscheinlich).

3.3 Wählen Sie eine Formel, durch die folgende Wahrscheinlichkeitsverteilung simuliert wird:
0 mit einer Wahrscheinlichkeit von 70 %
1 mit einer Wahrscheinlichkeit von 30 %

4. Beweisen Sie durch Einsetzen aller möglichen Kombinationen von TRUE und FALSE für A und B, daß die folgenden booleschen Ausdrücke äquivalent sind (Gesetze von de Morgan):
NOT(A AND B) entspricht NOT(A) OR NOT(B)
NOT(A OR B) entspricht NOT(A) AND NOT(B)

2.7 Deklaration von Konstanten

Syntax der Konstantendeklaration

Oft gibt es gewisse Werte in einem Programm, die während der gesamten Laufzeit des Programmes nicht verändert werden. Für diese **Konstanten** kann man in Pascal Namen vergeben. Wie die Variablendeklaration muß die Konstantendeklaration im Vereinbarungsteil erfolgen. Während der *report* verlangt, daß der Konstanten-Vereinbarungsteil vor dem Variablen-Vereinbarungsteil steht, ist in *Pascal 2.0* die Reihenfolge dieser Teile irrelevant.

```
PROGRAM KONSTANTEN(OUTPUT);
  CONST FAKTOR1 = 1.7453292E-2; (* = PI / 180    *)
        FAKTOR2 = 57.29577951;  (* = 1 / FAKTOR1 *)
        CLEARSCREEN = #147;
        RESETWINDOW = #19 #19;
        TITEL       = 'Dies ist ein Beispielstring';
        XLO  =  5;              YLO =  0;
        XRU  = 25;              YRU = 15;
BEGIN
  WRITELN(CLEARSCREEN, TITEL);
  WRITELN(TITEL); WRITELN(TITEL); WRITELN(TITEL);
  WINDOW (XLO, YLO, XRU, YRU, 1);
  WRITELN(TITEL); WRITELN(TITEL);
  WRITE(RESETWINDOW);
END.
```

Bild 10: Ein Programm mit Konstanten

Eine Konstantendeklaration wird durch das Wortsymbol CONST eingeleitet. Jedem Namen wird nach einem Gleichheitszeichen (**nicht** »:=«) ein Wert zugeordnet. Dabei können beliebige Konstanten verwendet werden. Der Typ der Konstanten bestimmt auch den Typ des Konstantennamens. So ist YLU eine Konstante vom Typ INTEGER, während FAKTOR1 vom Typ REAL ist.

Nach dem Gleichheitszeichen darf nur eine Konstante (eventuell mit einem Vorzeichen) folgen. Andere Ausdrücke sind nicht erlaubt. Die Werte der Konstanten können somit bereits zum Übersetzungszeitpunkt bestimmt werden.

```
CONST FAKTOR3 = - FAKTOR1;          (* zulässig *)
      FAKTOR2 = 1 / FAKTOR1;        (* nicht erlaubt *)
```

Konstanten sorgen für klare Programme

Es gibt viele Gründe für die Verwendung von Konstantennamen: Das wichtigste Argument ist die Tatsache, daß der Wert nur an einer einzigen Stelle im Quelltext auftritt. Damit können spätere Programmänderungen **konsistent** durch die Modifikation der Konstantendeklaration durchgeführt werden. Beispiele für solche Änderungen finden Sie unter anderem in Abschnitt 2.9 bei der Deklaration von Array-Variablen. Des weiteren erhöhen Konstantennamen den Dokumentationswert eines Programmes erheblich:

```
IF ZAEHLER = ENDWERT THEN ....
```

ist wesentlich aussagekräftiger als

```
IF ZAEHLER = 342 THEN
```

Stringkonstanten mit Kontrollzeichen

Insbesondere bei Stringkonstanten fällt außerdem die Speicherplatzeinsparung durch die Verwendung von Konstanten ins Gewicht. Die Stringkonstante muß nämlich nur genau einmal im Programmspeicher vorhanden sein.

Das Beispielprogramm in Bild 10 zeigt die Verwendung von Steuersequenzen in *Pascal 2.0* zur Bildschirmausgabe: Zunächst werden einige Zeilen Text gedruckt. Anschließend wird über dem gedruckten Text ein verkleinertes Bildschirmfenster (*window*) definiert, in dem wiederum einige Textzeilen gedruckt werden. Vor dem Ende des Programmes wird mit dem String RESETWINDOW die Fensterdefinition wieder aufgehoben, so daß nachfolgende Aus- und Eingaben wieder auf dem gesamten Bildschirm möglich sind. Die Parameter der vordefinierten Prozedur WINDOW sind in Abschnitt 4.4.4.10 erklärt.

2.8 Kontrollstrukturen

Kontrollstrukturen in BASIC zum Vergleich

Bisher haben Sie nur lineare Programme kennengelernt, das sind Programme, in denen die Anweisungen genau in der Reihenfolge ausgeführt werden, in der sie im Programmtext stehen. Eine entscheidende Fähigkeit von Computern ist jedoch gerade die Möglichkeit, Anweisungen zu wiederholen oder nur in Abhängigkeit von Bedingungen auszuführen, die während des Programmes geprüft werden. In BASIC wird diese Programmablaufkontrolle durch (bedingte) Sprünge im Programm erreicht (IF, GOTO, GOSUB, FOR ... NEXT, ON ... GOTO, ON ... GOSUB).

In diesem Abschnitt werden die entsprechenden Kontrollstrukturen in Pascal vorgestellt. Bedingungen und Wiederholungen werden durch sogenannte zusammengesetzte Anweisungen gebildet, die dem Programm eine **Blockstruktur** geben.

Strukturierte Programmierung durch Blockstruktur der Anweisungen

Diese Blockstruktur ist die Grundlage für ein fundamentales Prinzip der Strukturierten Programmierung: Der statische Programmtext zeigt bereits die dynamische Struktur (z.B. Schleifen) des Programmes. Ein Programm besteht dementsprechend aus Blöcken, die nur einen Eingang und einen Ausgang besitzen. Ein Zusammenstellen von Blöcken ist nur nach festen Regeln möglich, wobei immer wieder Blöcke mit nur einem Eingang und Ausgang gebildet werden.
Das mag Sie im Augenblick noch recht abstrakt anmuten, jedoch werden die Abbildungen in diesem Abschnitt den Zusammenhang zwischen der geschachtelten Einrückung des Programmtextes und der dadurch symbolisierten Schachtelung der **Anweisungen** im Programm verdeutlichen.
Ein Lerneffekt bei der Lektüre des zweiten Kapitels sollte sein, daß Sie durch *Hinschauen* die Struktur eines fremden Pascal-Programmes erkennen, ohne erst durch kleinliche Analyse jeder Anweisung zum Kern des Programmes vorzudringen.
Die elementaren Bausteine eines Programmes sind die **einfachen Anweisungen**. Beispiele für einfache Anweisungen kennen Sie bereits aus den Abschnitten 2.4 und 2.5. Es sind dies

Einfache Anweisungen werden zu zusammengesetzten Anweisungen zusammengefaßt

1. Zuweisungen von Werten eines Ausdruckes an eine Variable.
2. Aufrufe von vordefinierten oder benutzerdefinierten Prozeduren. So ist z.B. die Anweisung WRITE(7) ein Aufruf der Standardprozedur WRITE mit dem Parameter 7.
3. Sprunganweisungen mit dem Befehl GOTO innerhalb des Programmes.

Die Sprunganweisungen sind wahre Exoten in Pascal-Programmen, die in manchen Lehrbüchern (aus didaktischen Gründen) nicht einmal behandelt werden. Somit besteht ein Programm ausschließlich aus Zuweisungen und Prozeduraufrufen, die in geeigneter Reihenfolge aufgerufen werden. Diese »geeignete Reihenfolge« wird durch die sogenannten **zusammengesetzten Anweisungen** definiert, die nun jeweils mit Beispielen vorgestellt werden.

2.8.1 Anweisungsfolgen

BEGIN und END = Klammer-Anweisungen

Falls es nicht durch Kontrollstrukturen anders definiert wird, werden die Befehle eines Programmes strikt sequentiell ausgeführt. Wie Sie später erfahren werden, ist es oft nötig, eine Folge von Anweisungen zu einer **Einheit** zusammenzufassen. Betrachten Sie zum Beispiel die folgende **Anweisungsfolge**:

```
BEGIN
  READLN(A, B);
  A:= SQRT(A*A + B*B);
  WRITELN('BETRAG =', A)
END
```

Sie besteht aus zwei Prozeduraufrufen (READLN und WRITELN) und einer Zuweisung. Diese Anweisungen werden durch Semikola voneinander getrennt und mit den reservierten Wortsymbolen BEGIN und END *geklammert*. Offensichtlich besteht also der Anweisungsteil eines Programmes aus einer einzigen Anweisungsfolge. Die folgende Abbildung zeigt die allgemeine Struktur einer Anweisungsfolge:

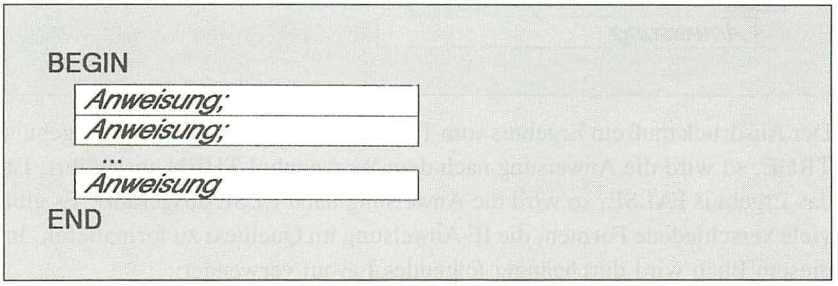


Bild 11: Struktur einer Anweisungsfolge

Die Klammerung mit BEGIN und END wird später dazu verwendet werden, um in zusammengesetzten Anweisungen, bei denen eine einzelne Anweisung erwartet wird, eine ganze Anweisungsfolge einzusetzen. Dies wird in der Abbildung durch den umfassenden Rahmen angedeutet.

Das Semikolon in Anweisungsfolgen

Noch ein Wort zu den Semikola: Ein Semikolon **trennt** Anweisungen. Deshalb ist kein Semikolon vor dem abschließenden END erforderlich. Setzen Sie auch dort ein Semikolon,

```
BEGIN
  READLN(A, B);
  A:= SQRT(A*A + B*B);
  WRITELN('BETRAG =', A);
END
```

Leeranweisungen

so erwartet der Compiler eine Anweisung. Um diesen Fall nicht als Fehler zu behandeln, gibt es in Pascal die Leeranweisung, die aus keinem Befehl besteht. Diese mysteriöse Anweisung tritt auch in den folgenden (korrekten) Anweisungsfolgen auf:

```
BEGIN END            (1 Leeranweisung)
BEGIN A:=B; END      (1 Leeranweisung)
BEGIN ; ; END        (3 Leeranweisungen)
```

Deshalb besitzt das Syntaxdiagramm ANWEISUNG in Anhang A auch einen »leeren Pfad«, auf dem kein Symbol steht.

2.8.2 Bedingte Anweisungen

Syntax der IF-Anweisung

Eine bedingte Anweisung (IF-Anweisung) hat die folgende Form:

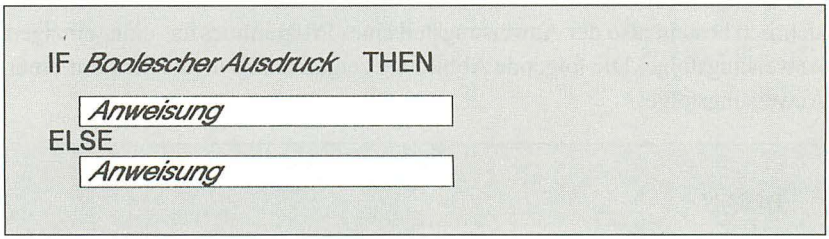


Bild 12: Struktur der IF-Anweisung

Der Ausdruck muß ein Ergebnis vom Typ BOOLEAN liefern. Ist das Ergebnis TRUE, so wird die Anweisung nach dem Wortsymbol THEN ausgeführt. Ist das Ergebnis FALSE, so wird die Anweisung nach ELSE ausgeführt. Es gibt viele verschiedene Formen, die IF-Anweisung im Quelltext zu formatieren. In diesem Buch wird durchgängig folgendes Layout verwendet:

Zweiseitige Auswahl

```
IF KONTO>=0 THEN
  WRITELN(KONTO :8, ' DM Guthaben')
ELSE
  WRITELN(KONTO :8, ' DM Schulden')
```

Wie Sie bereits wissen, ist dem Compiler nur die Symbolfolge und nicht ihre Formatierung wichtig, so daß er auch diese Einrückung akzeptiert:

```
PROGRAM MAXIMUM (INPUT,OUTPUT);
  VAR A,B,MAX: INTEGER;
BEGIN
  READLN(A,B);
  IF A>B THEN MAX:= A
        ELSE MAX:= B;
  WRITELN('Das Maximum von ',A, ' und ',B,' ist ', MAX);
END.
```

Einseitige Auswahl

Bei einer anderen Form der IF-Anweisung ist kein ELSE-Zweig vorgesehen. Nur wenn die Bedingung des booleschen Ausdruckes zutrifft, wird die Anweisung nach THEN ausgeführt:

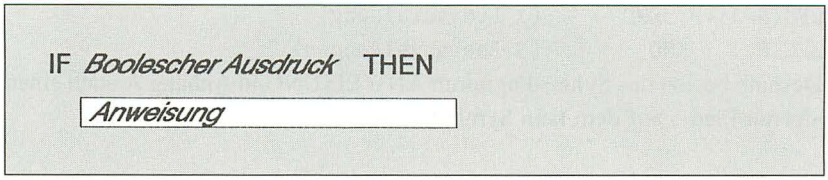


Bild 13: Einseitige Auswahl

Somit läßt sich das Maximum zweier Zahlen auch durch Austauschen bestimmen:

```
PROGRAM MAXSWAP (INPUT,OUTPUT);
  VAR A,B,H : INTEGER;
BEGIN
  READLN(A,B);
  IF A<B THEN
    BEGIN H:=A; A:=B; B:=H END;
  (* jetzt ist A>=B *)
  WRITELN(A, ' >= ', B);
END.
```

Anweisungsfolgen nach THEN und ELSE

Dieses Beispiel zeigt auch, wie man statt einer einzelnen Anweisung eine ganze Anweisungsfolge durch eine Bedingung kontrolliert. Durch die Klammerung mit BEGIN und END erkennt der Compiler, welche Anweisungen der THEN-Teil umfaßt. Fehlen diese Klammern, so wird das Programm nicht korrekt übersetzt:

Falsch!

```
IF A>B THEN
  MAX:= A; MIN:= B
ELSE
  MAX:= B; MIN:= A
```

Nach der oben angegebenen Syntax der IF-Anweisung muß nach THEN und ELSE eine einzelne Anweisung folgen. Deshalb kann der Compiler das Wortsymbol ELSE nicht verarbeiten, was Sie spätestens an der Fehlermeldung

```
175 ERROR IN STATEMENT SEQUENCE
```

erkennen. Um die Struktur, die durch die Einrückung beschrieben wird, auch korrekt in Pascal zu formulieren, muß man also die Anweisungen MAX:=A; MIN:=B zu einer Anweisungsfolge zusammenfassen:

Richtig!

```
IF A>B THEN
  BEGIN MAX:= A; MIN:= B; END
ELSE
  BEGIN MAX:= B; MIN:= A; END
```

Kein Semikolon vor ELSE

Auf einen weiteren Fallstrick müssen Sie noch achten: Hätten Sie vor dem Wortsymbol ELSE ein Semikolon gesetzt,

Falsch!

```
IF A>B THEN
  BEGIN MAX:= A; MIN:= B END;
ELSE
  BEGIN MAX:= B; MIN:= A END
```

so würde der Compiler eine einseitige Auswahl wie in Bild 13 erkennen, und wiederum das ELSE mit der Fehlermeldung 175 markieren.

Schachtelung von IF-Anweisungen

Am Beispiel der bedingten Anweisungen läßt sich bereits das Prinzip der **Schachtelung** von Anweisungen (Blöcken) erklären: Eine IF-Anweisung ist eine zusammengesetzte Anweisung. Dadurch werden die Anweisungen nach THEN und ELSE zusammen mit der IF-Abfrage zu **einer einzigen** Anweisung

zusammengefaßt. Das sollen auch die äußeren Rahmen in Bild 12 und Bild 13 verdeutlichen. Deshalb kann eine IF-Anweisung selbst als Teil einer anderen IF-Anweisung auftreten. Um die kleine Sammlung der Programme zur Bestimmung von Maxima und Minima zu komplettieren, folgt als Beispiel die Berechnung des Maximums dreier ganzer Zahlen:

```
IF A>B THEN
  IF A>C THEN
    MAX:=A
  ELSE
    MAX:=C
ELSE
  IF B>C THEN
    MAX:=B
  ELSE
    MAX:=C
```

Bild 14: Geschachtelte Anweisungen

Im äußeren Block wird geprüft, ob A > B ist. Ist dies der Fall (A > B ist TRUE), so wird der erste geschachtelte Block ausgeführt, in dem der Vergleich A > C stattfindet. Je nach dem Ergebnis dieser zweiten Abfrage wird MAX:=A oder MAX:=C ausgeführt. War jedoch bereits im äußeren Block A < =C, so wird die andere geschachtelte IF-Anweisung (B > C) ausgeführt.

Probleme mit ELSE-Schachtelungen

Das obige Beispiel ist noch sehr übersichtlich, jedoch können Schachtelungen auftreten, bei denen die Vielzahl von IF, THEN und ELSE Verwirrung stiften kann:

```
PROGRAM UEBLESCHACHTEL(INPUT,OUTPUT);
  VAR X: INTEGER;
BEGIN
  READLN(X);
  IF X>0 THEN
    IF ODD(X) THEN
      WRITELN('X ist größer als 0 und ungerade')
ELSE
  WRITELN('X ist kleiner oder gleich 0');
END.
```

Falsch!

Sie sollten sich ruhig die Zeit nehmen, dieses Programm einzutippen und zu testen. Dann werden Sie nämlich feststellen, daß der Compiler ein Programm erzeugt, in dem der Text »X ist kleiner oder gleich 0« immer dann ausgegeben wird, wenn X größer als Null und gerade ist.

Was ist also passiert? Offenbar wurde das ELSE dem »näherliegenden« IF ODD(X) zugeordnet. In Pascal gilt nämlich folgende Regel:

Im Zweifelsfall wird bei geschachtelten IF-THEN-ELSE-Anweisungen jedes ELSE dem nächsten IF zugeordnet, zu dem noch kein ELSE gehört.
Bevor Sie sich jedoch Sorgen über diese Spezialregel machen, sei eine einfache Methode zur Vermeidung von Zweideutigkeiten vorgestellt: Es genügt nämlich, die gewünschte Zuordnung von IF und ELSE durch eine Klammerung mit BEGIN und END zu erzwingen:

Richtig!

```
PROGRAM GUTESCHACHTEL(INPUT,OUTPUT);
  VAR X: INTEGER;
BEGIN
 READLN(X);
 IF X>0 THEN
   BEGIN
     IF ODD(X) THEN
       WRITELN('X ist größer als 0 und ungerade')
    END
 ELSE
 WRITELN('X ist kleiner oder gleich 0');
END.
```

2.8.3 Fallunterscheidung

Vereinfachung komplexer IF-Anweisungen mit der CASE-Anweisung

In manchen Fällen muß man in Abhängigkeit eines einzigen Wertes unterschiedliche Operationen vornehmen. Mit einer IF-Anweisung erhält man dann Konstrukte der folgenden Form:

```
IF CH= '+' THEN ERG:= A + B
ELSE
  IF CH = '-' THEN ERG:= A - B
  ELSE
    IF CH = '*' THEN ERG:= A * B
    ELSE
      IF CH = '/' THEN ERG:= A / B
      ELSE
        WRITELN('Die Operation ', CH, ' ist nicht erlaubt');
```

Abgesehen davon, daß man bei einer solchen Einrückung bald den rechten Bildschirmrand erreichen wird, benötigen die vielen identischen Vergleiche unnötig viel Code. Für Anwendungen wie diese existiert in Pascal die Fallunterscheidung (CASE-Anweisung):

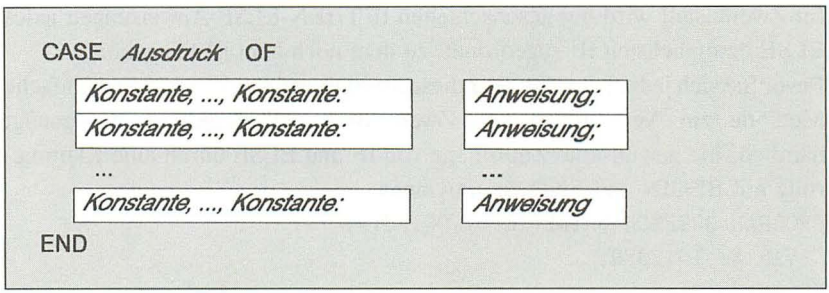


Bild 15: Struktur der CASE-Anweisung

Semantik der CASE-Anweisung

Eine Fallunterscheidung wird folgendermaßen ausgeführt. Zunächst wird der Ausdruck ausgewertet. Er muß einen Wert eines skalaren Typs (z.B. CHAR, INTEGER, aber nicht REAL) liefern. Von den nachfolgenden Anweisungen wird nur diejenige ausgeführt, die den Wert des Ausdruckes in ihrer Konstantenliste enthält. Natürlich müssen die Fallmarken denselben Typ wie der Ausdruck besitzen. Kommt der Wert des Ausdruckes in keiner Fallmarke vor, so erfolgt ein Programmabbruch mit der Fehlermeldung:

```
NO LABEL IN CASE ERROR
```

Erweiterungen in Pascal 2.0

In *Pascal 2.0* ist die Syntax der CASE-Anweisung dahingehend erweitert worden, daß am Ende der Fallmarken (vor dem abschließenden END) noch das Wortsymbol ELSE, gefolgt von einer Anweisung, stehen darf. Diese Anweisung wird ausgeführt, wenn der Wert des Ausdruckes mit keiner Fallmarke übereinstimmt. Das folgende Beispielprogramm zeigt eine Anwendung der CASE-Anweisung zur Simulation eines einfachen Taschenrechners: Der Benutzer gibt zwei reelle Zahlen A und B sowie ein Operationszeichen ein. Das Ergebnis der Verknüpfung der Zahlen wird vom Computer ausgegeben:

Ein Taschenrechner in Pascal

Eingabe: * 300 40

Ausgabe: 3.00000000E+02 * 4.00000000E+01 = 1.2000000E+04

```
PROGRAM COMPUTER (INPUT,OUTPUT);
  VAR OPERATION  : CHAR;
      A, B, ERG  : REAL;
      OK         : BOOLEAN;
BEGIN
  READLN(OPERATION, A, B);
  OK:= TRUE;              (* Noch kein Fehler aufgetreten *)
  CASE OPERATION OF
   '.', '*' : BEGIN ERG:= A * B; END;
   '/'      : BEGIN
                IF B = 0.0 THEN
               BEGIN
                 OK:= FALSE;
                 WRITELN('Keine Division durch Null!');
```

```
            END
          ELSE
            ERG:= A / B;
          END;
  '+'     : ERG:= A + B;
  '-'     : ERG:= A - B
  ELSE     BEGIN
            OK:= FALSE;
            WRITELN('Operation ',OPERATION, ' unbekannt');
           END;
 END; (* CASE *)
 IF OK THEN
   WRITELN(A, ' ', OPERATION, ' ', B, ' = ', ERG);
END.
```

Boolesche Variablen zur Wahrung der Blockstruktur

Dieses Programm zeigt auch eine Anwendung boolescher Variablen. Durch die Blockstruktur von Pascal-Programmen ist es nicht möglich, eine zusammengesetzte Anweisung »vorzeitig« zu verlassen und z.B. die Ausgabe des Ergebnisses zu »überspringen« (was wohl die übliche Realisierung in BASIC wäre). Statt dessen merkt man sich das Auftreten eines Fehlers, indem man die boolesche Variable OK auf FALSE (also *nicht ok*) setzt.

Wenn Sie sich noch an die grundsätzlichen Aussagen über die Blockstruktur in Pascal am Anfang des Kapitels 2.8 erinnern, so können Sie sich vielleicht jetzt etwas mehr darunter vorstellen, was es bedeutet, daß ein Block nur **einen** Eingang und **einen** Ausgang besitzen darf: Egal wie kompliziert sich der Programmfluß innerhalb der CASE-Anweisung verästelt, am Ende der Ausführung wird auf jeden Fall die Anweisung IF OK THEN ... ausgeführt.

Übrigens ist im Beispiel die Klammerung mit BEGIN und END bei den Fallmarken '*' und '.' nicht erforderlich, jedoch erleichtern solche redundanten Anweisungsfolgen oft die Erweiterung eines Programmes um zusätzliche Anweisungen.

Cursorsteuerung mit der CASE-Anweisung

Eine häufige Anwendung der CASE-Anweisung besteht in der Auswertung von Tastatureingaben. Das nächste Programmstück zeigt eine simulierte Cursorsteuerung: Mit den Cursortasten läßt sich ein Stern auf dem Bildschirm bewegen. Durch die Anwendung der Operation MOD erscheint der Cursor beim Überschreiten des Bildschirmrandes auf der gegenüberliegenden Seite wieder. Durch die Betätigung der Return-Taste wird das Programm beendet. Im Vorgriff auf Abschnitt 2.8.5 wird bereits die REPEAT-Anweisung benutzt:

```
PROGRAM MOVECURSOR(INPUT,OUTPUT);
  CONST CRSRUP    = 145;(* Tastaturcodes:   *)
        CRSRDOWN  = 17;
        CRSRRIGHT = 29;
```

```
        CRSRLEFT  = 157;
        RETURN    = 13;
        WIDTH     = 39; (* Bildschirmbreite *)
        HEIGHT    = 24; (* Bildschirmhoehe  *)
        (* Breite und Höhe für 40-Zeichen Bildschirm *)
  VAR   KEY: CHAR;
        X,Y: INTEGER;
BEGIN
  X:= WIDTH  DIV 2; (* Cursor auf Bildschirmmitte *)
  Y:= HEIGHT DIV 2;
  REPEAT
    DISPLAY(1,X,Y,'*');          (* Cursor an  *)
    REPEAT GETKEY(KEY) UNTIL ORD(KEY)<>0;
    DISPLAY(1,X,Y,' ');          (* Cursor aus *)
    CASE ORD(KEY) OF
      CRSRUP   : IF Y=0 THEN Y:= HEIGHT
                 ELSE Y:= Y-1;
      CRSRDOWN : Y:= (Y+1) MOD HEIGHT;
      CRSRLEFT : IF X=0 THEN X:= WIDTH
                 ELSE X:= X-1;
      CRSRRIGHT: X:= (X+1) MOD WIDTH
      RETURN   : BEGIN (* Leeranweisung *) END
      ELSE WRITE( #7)            (* piepsen *)
    END;
  UNTIL ORD(KEY)=RETURN;
END.
```

Zum Schluß sei darauf hingewiesen, daß manche Compiler (nicht *Pascal 2.0*) Fallmarken aus einem zusammenhängenden Wertebereich erwarten. Sie würden für folgendes Beispiel einen sehr ungünstigen Code erzeugen:

```
CASE X OF
 2   : BEGIN ... END;
 200 : BEGIN ... END;
 2000: BEGIN ... END;
END
```

2.8.4 WHILE-Anweisung

Wiederholung eines Blocks

Möchte man eine Anweisung (einen Block) wiederholt ausführen, so gibt es dafür in Pascal drei verschiedene zusammengesetzte Anweisungen, die jeweils unterschiedliche Kontrollstrukturen bilden. Die Unterschiede bestehen in der Form, in der die Bedingung formuliert wird, unter der die Anweisung wieder-

holt wird. Zunächst wird die am häufigsten benutzte Wiederholungsanweisung, die WHILE-Anweisung, beschrieben.

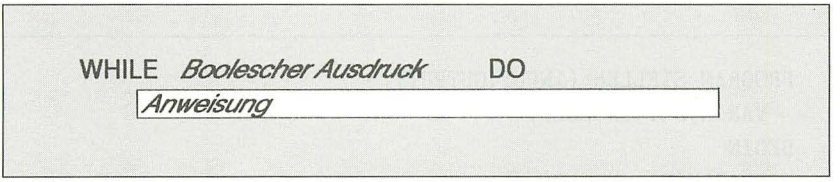


Bild 16: Struktur der WHILE-Anweisung

```
PROGRAM WHILEDEMO (INPUT,OUTPUT);
  VAR VON,BIS,X: INTEGER;
BEGIN
  WRITE('VON BIS :'); READLN(VON,BIS);
  X:= VON;
  WHILE X<=BIS DO
    BEGIN WRITE(X); X:= X+1; END;
END.
```

Bild 17: Eine einfache WHILE-Schleife

Semantik der WHILE-Anweisung

Eine WHILE-Anweisung wird folgendermaßen ausgeführt:

1. Der boolesche Ausdruck wird ausgewertet. Ist die Bedingung nicht erfüllt (FALSE), so wird die gesamte WHILE-Anweisung beendet.
2. War die Bedingung erfüllt, das Ergebnis des Ausdruckes also TRUE, so wird die Anweisung nach dem Wortsymbol DO ausgeführt.
3. Anschließend wird die Programmausführung bei (1) fortgesetzt.

In Bild 17 wird also der Wert der Variablen X so lange mit WRITE ausgegeben und um 1 erhöht, bis X größer als die Variable BIS wird. Beispiele für die Ausgaben des Programmes zeigen eine wichtige Eigenschaft der WHILE-Anweisung:

```
VON BIS: 1   5
1 2 3 4 5
VON BIS: 1   2
1 2
VON BIS: 1   1
1
VON BIS: 1   0
```

Prüfung des booleschen Ausdrucks vor der Schleife

Ist also wie bei der letzten Eingabe die Bedingung bereits beim Eintritt in die Schleife nicht erfüllt, wird die Anweisung **überhaupt nicht** ausgeführt. In diesem Beispiel wurde wieder eine mit BEGIN und END geschachtelte Anweisungsfolge verwendet, um mehr als eine Anweisung nach dem Wortsymbol DO zu wiederholen.

Das nächste Programmbeispiel in Bild 18 berechnet die Anzahl der Stellen einer ganzen Zahl. Die Idee besteht darin, zu zählen, wie oft man die Zahl X nach *rechts schieben* kann, bis alle Ziffern hinter dem Komma stehen:

```
PROGRAM STELLEN (INPUT,OUTPUT);
  VAR X,N : INTEGER;
BEGIN
  READLN(X); N:= 0;
  (* N zählt die Divisionen *)
  WHILE X<>0 DO
    BEGIN
      X:= X DIV 10; N:= N+1
    END;
  WRITELN('Anzahl der Stellen:', N);
END.
```

Bild 18: Dezimalstellen zählen

Behandlung von Sonderfällen

Wichtig ist dabei die Tatsache, daß die Prüfung des Ausdruckes (X< >0) **vor** der Ausführung der Anweisungen (X:= X DIV 10; N:= N+1) erfolgt. Deshalb wird für die Eingabe der Zahl X=0 die Schleife überhaupt nicht durchlaufen, weshalb das Programm für diese Eingabe das Ergebnis N=0 liefert. Anzumerken ist noch, daß das Programm auch für negative Zahlen korrekt arbeitet.
Nach der ausführlichen Diskussion im letzten Abschnitt ist Ihnen sicher auch klar, warum im Programm STELLEN nach dem Wortsymbol DO eine Anweisungsfolge (BEGIN ... END) steht. Ist dies nicht der Fall, sollten Sie das Programm probeweise ohne die Klammerung mit BEGIN und END übersetzen und testen. Wenn Sie anschließend die beiden letzten Kapitel noch einmal lesen, werden Sie die Bedeutung der Anweisungsfolge zur Bildung von Anweisungsblöcken erkennen.

Geschachtelte WHILE-Anweisungen

In Bild 19 werden zwei geschachtelte WHILE-Anweisungen verwendet, um ein Netz im Grafikmodus zu zeichnen. In *Pascal 2.0* muß bei der Verwendung des Grafikmodus der Name GRAPHIC im Programmkopf aufgeführt werden. Nachdem mit der Anweisung GRAPHIC der Grafikbildschirm eingeschaltet wurde, werden N1-Punkte mit N2-Punkten durch je eine Linie verbunden. Dabei zählt die Variable P1 in der *äußeren* WHILE-Anweisung von 1 bis N1, während die Variable P2 in der *inneren* WHILE-Anweisung von 1 bis N2 zählt. Die Prozedur DRAW (siehe auch Kapitel 4.4.4.9) verbindet nun jeweils zwei Punkte (P1 und P2) in der Zeichenfarbe (1). Durch die Wahl der Faktoren (20, 0, 25, 10) werden zwei Geraden auf dem Bildschirm definiert, auf denen jeweils die Start- und Endpunkte der Linien in gleichem Abstand liegen. Im Programm werden die Punkte nun nach folgendem Schema verbunden:

P1 auf Gerade 1	mit	P2 auf Gerade 2
1		1, 2, 3, 4, 5, 6, 7, 8
2		1, 2, 3, 4, 5, 6, 7, 8
...		
8		1, 2, 3, 4, 5, 6, 7, 8

```
PROGRAM NETZ (INPUT,OUTPUT,GRAPHIC);
  CONST N1 = 8;
        N2 = 8;
  VAR P1,P2: INTEGER;
BEGIN
  GRAPHIC(1,1);
  P1:= 1;
  WHILE P1<=N1 DO
   BEGIN
     P2:= 1;
     WHILE P2<=N2 DO
       BEGIN                    (* Zeichne eine Linie: *)
         DRAW(1,                (* Farbe               *)
              P1*40, P1*0,      (* Startpunkt          *)
              P2*0, P2*20);     (* Endpunkt            *)
         P2:= P2+1;
       END;
     P1:= P1+1;
   END;
  REPEAT UNTIL KEYPRESSED;   (* Warte auf Tastatureingabe *)
  GRAPHIC(0);
END.
```

Bild 19: Geschachtelte WHILE-Anweisungen

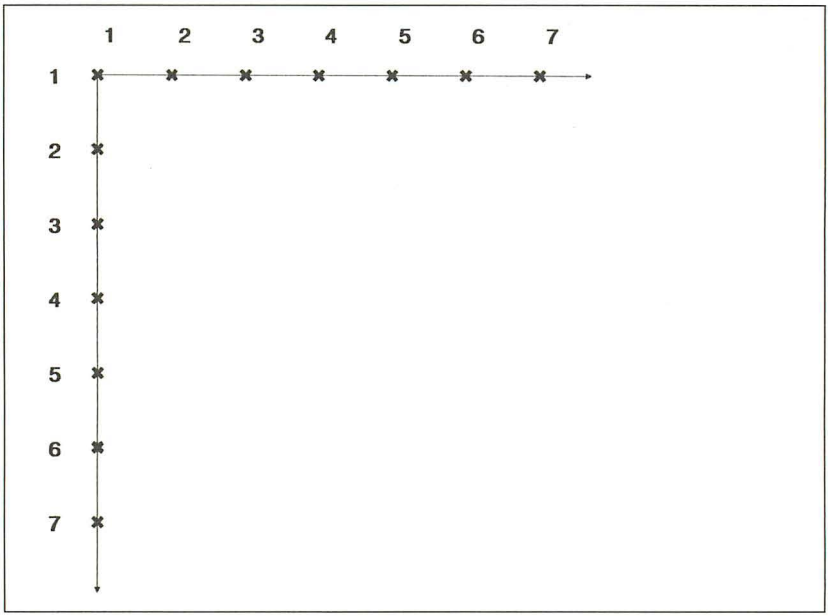


Bild 20: Lage der Linienendpunkte

Insgesamt werden also N1 * N2 = 64 Linien gezeichnet. Durch eine Änderung der Faktoren bei der DRAW-Prozedur und die Variation der Punkteanzahlen (N1 und N2) können Sie verschiedene Effekte erzielen. Probieren Sie doch zum Beispiel

```
N1 = 10; N2 = 10
DRAW(1,
     P1 * 32, 100,
     160     , P2 *20)
```

Potenzieren mit der WHILE-Anweisung

WHILE-Anweisungen nennt man auch *pre check loops* oder abweisende Schleifen. Durch die Eigenschaft einer WHILE-Anweisung, die Schleife auch keinmal auszuführen, spart man oft notwendige Sonderbehandlungen. Das letzte Beispielprogramm dieses Abschnittes berechnet für zwei natürliche Zahlen N und K den Wert E = N hoch K.

```
PROGRAM HOCH (INPUT, OUTPUT);
  VAR E,N,I,K : INTEGER;
BEGIN
  READLN(N,K);
  E:= 1; I:=K;
  WHILE I>0 DO
   BEGIN E:= E*N; I:= I-1; END;
  WRITELN(N, '↑', K, '=', E);
END.
```

Dieses Programm berücksichtigt auch die Sonderfälle N=0 und K=0 korrekt. Es gilt nämlich:

N hoch 0 = 1 für alle N

0 hoch K = 0 für alle K< >0

Allgemeine Regeln für Schleifen

Auch für die REPEAT-Anweisung, die erst im nächsten Abschnitt vorgestellt wird, gelten die folgenden Regeln, die man bei der Programmierung von Wiederholungsanweisungen beachten sollte:

1. Bei der Wiederholung muß die Schleife irgendwann beendet werden. Deshalb muß man sich während der Berechnung dem Ziel nähern. Folgendes Programmstück ist also auf jeden Fall sinnos, da der Wert des booleschen Ausdruckes (I< >0) innerhalb der Schleife überhaupt nicht verändert wird:
   ```
   WHILE I< >0 DO K:= K+1
   ```
2. Während jeder Ausführung der Wiederholung müssen gewisse Bedingungen erhalten bleiben. Diese Bedingungen bezeichnet man auch als *Schleifen-Invarianten*. Es ist keine schlechte Idee, diese Invarianten durch Kommentare zu verdeutlichen.
   ```
   E:=1; I:= K;
   WHILE I>0 DO
    (* E = X hoch (K-I) *)
    BEGIN E:= E*N; I:= I-1 END
   ```
   Am Ende der Schleife ist also I=0, und damit besitzt E den gewünschten Wert.
3. Grundsätzlich sollte man beim Entwurf einer Wiederholung den *Sonderfällen* besondere Aufmerksamkeit schenken, um logische Fehler nicht erst im fertigen Programm bei seltenen Eingaben zu finden.

2.8.5 REPEAT-Anweisung

Seltener als die WHILE-Anweisung wird die REPEAT-Anweisung benutzt. Sie hat die in Bild 21 angegebene Struktur:

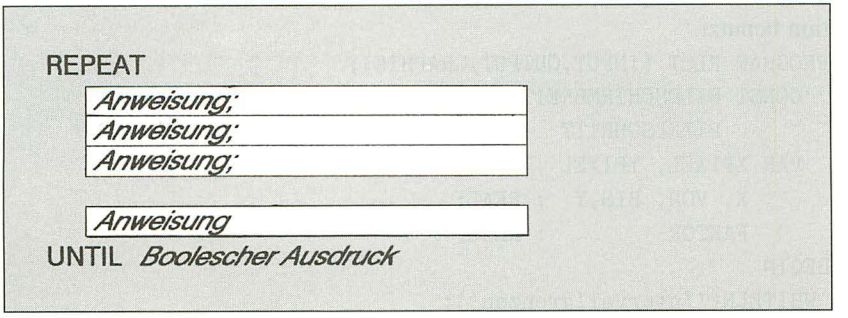


Bild 21: REPEAT-Anweisung

Semantik der REPEAT-Anweisung

Der wesentliche Unterschied zur WHILE-Anweisung ist die Tatsache, daß die Anweisungsfolge mindestens einmal ausgeführt wird, da eine REPEAT-Anweisung folgendermaßen ausgeführt wird:

1. Die Anweisungen zwischen den Wortsymbolen REPEAT und UNTIL werden in der angegebenen Reihenfolge ausgeführt.
2. Anschließend wird die Bedingung nach dem Symbol UNTIL ausgewertet. Ist die Bedingung erfüllt (TRUE), so wird die Ausführung der REPEAT-Anweisung beendet. Ist das Ergebnis des booleschen Ausdruckes FALSE, so wird die Ausführung bei (1) fortgesetzt.

Einfache Beispiele

Im Gegensatz zur WHILE-Anweisung kann damit der boolesche Ausdruck Variablen enthalten, die erst innerhalb der Anweisungsfolge berechnet werden.

```
PROGRAM SIMPLEREPEAT(INPUT,OUTPUT);
   VAR CH: CHAR;
BEGIN
  REPEAT
    WRITE('Alles klar? (J,N) ');
    READLN(CH)
  UNTIL (CH='J') OR (CH='N');
  IF CH ='J' THEN
    WRITELN('Na prima!')
  ELSE
    WRITELN('Ist doch gar nicht schwer!');
END.
```

Eine beliebte Anwendung der REPEAT-Anweisung besteht nur aus einer Abbruchbedingung ohne Anweisungen zwischen REPEAT und UNTIL:

```
REPEAT UNTIL KEYPRESSED;
```

Die in *Pascal 2.0* vordefinierte Funktion KEYPRESSED liefert den Wert TRUE, falls der Benutzer eine Taste betätigt hat. Somit wird die (leere) REPEAT-Schleife bis zur Betätigung einer Taste durchlaufen. In Bild 19 wurde diese Schleife zum Beispiel verwendet, um vor dem Löschen des Grafikbildschirms auf einen Tastendruck des Benutzers zu warten.

Ausgabe einer Funktion am Bildschirm

Im nächsten Beispiel wird die REPEAT-Anweisung zum Zeichnen einer Funktion benutzt.

```
PROGRAM PLOT (INPUT,OUTPUT,GRAPHIC);
  CONST BILDSCHIRMBREITE = 319;
        PIXELSCHRITT     = 3;
  VAR XPIXEL, YPIXEL : INTEGER;
      X, VON, BIS,Y  : REAL;
      FAKTOR         : REAL;
BEGIN
 WRITELN('Intervallgrenzen');
```

```
    WRITE('von :'); READLN(VON);
    WRITE('bis :'); READLN(BIS);
    FAKTOR:= (BIS-VON) / BILDSCHIRMBREITE;
    GRAPHIC(1,1); COLOR(1,1);          (* schwarze Farbe *)
    XPIXEL:= 0;
    REPEAT
      (* Berechne X-Wert aus XPIXEL: *)
      X:= VON + XPIXEL * FAKTOR;
      Y:= SIN(X) + 2 * COS(3*X);
      YPIXEL:= ROUND( Y*40) + 100;      (* Y Werte skalieren *)
      IF XPIXEL = 0 THEN                (* 1. Punkt *)
        LOCATE(XPIXEL,YPIXEL)
      ELSE
        DRAW(1,,XPIXEL,YPIXEL);         (* ziehe Linie *)
      XPIXEL:= XPIXEL + PIXELSCHRITT;
  UNTIL XPIXEL>BILDSCHIRMBREITE;
  REPEAT UNTIL KEYPRESSED;
  GRAPHIC(0);
END.
```

Erläuterungen zum Programm PLOT

Das Programm läßt sich dreiteilen: Zunächst werden die Grenzen des darzustellenden Koordinatenbereichs der Funktion eingegeben. In der nachfolgenden REPEAT-Schleife wird die Funktion von links nach rechts in voller Bildschirmbreite ausgegeben. Abschließend wird vor dem Löschen des Bildschirms auf eine Tastatureingabe des Benutzers gewartet. Wählen Sie z.B. folgende Eingabewerte

```
VON : -10
BIS :  10
```

wird die Funktion

```
Y = SIN(X) + 2 * COS(3*X)
```

im Intervall [–10, 10] dargestellt. Innerhalb der REPEAT-Schleife wird mit den Prozeduren LOCATE und DRAW (siehe 4.4.4.9) eine zusammenhängende Linie gezogen. Dabei wird in jedem Durchlauf der Schleife ein Bildschirmkoordinatenpaar XPIXEL / YPIXEL berechnet. Mit der bedingten Anweisung IF XPIXEL=0 wird folgende Zeichenstrategie gewählt:

Der erste Bildschirmpunkt wird mit LOCATE als Anfangspunkt eines Linienzuges definiert. In jedem folgenden Durchlauf wird der neu berechnete Punkt mit dem zuletzt gemalten Punkt verbunden.

Während XPIXEL mit der Schrittweite PIXELSCHRITT von 0 bis BILDSCHIRMBREITE hochgezählt wird, muß YPIXEL gemäß der Funktion in jedem Durchlauf neu berechnet werden. Mit der Zuweisung

```
X:= VON + XPIXEL * FAKTOR
```

Koordinatentransformation mit Typanpassung

wird der Bereich der ganzzahligen Bildschirmkoordinaten von 0 bis BILDSCHIRMBREITE auf den Bereich der X-Koordinaten der Funktion (z.B. [–10,10]) abgebildet. Dabei nimmt der Compiler automatisch eine Umwandlung vom Typ INTEGER zum Typ REAL vor. Nachdem so die Koordinate X berechnet wurde, kann diese in die Funktion eingesetzt werden, um die Y-Koordinate Y zu berechnen. Schließlich muß diese reelle Zahl Y wieder auf den Bereich der darstellbaren ganzzahligen Y-Bildschirmkoordinaten (von 0 bis 199) umgeformt werden. Dies geschieht mit der Rundungsfunktion ROUND:

```
YPIXEL:= ROUND (40*Y) + 100
```

Zur Abwechslung können Sie auch die folgende Funktion im Bereich von –20 bis 20 darstellen:

```
Y:= 2 * SIN(X) / X
```

2.8.6 FOR-Anweisung

FOR-Anweisung für Zählschleifen

In BASIC bietet die FOR-Anweisung die beste Möglichkeit zur Strukturierung von Wiederholungen. In Pascal wird die FOR-Anweisung nur dann verwendet, wenn die Anzahl der Wiederholungen bereits **vor** dem Eintritt in die *Schleife* bekannt ist. Die Struktur der Anweisung ist wieder in einem Bild dargestellt:

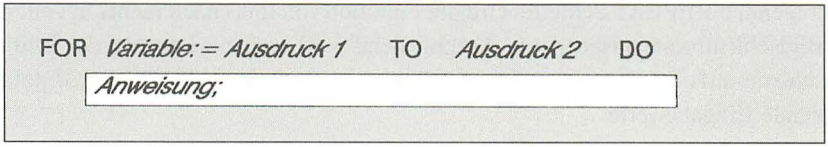


Bild 22: Struktur der FOR-Anweisung

Semantik der FOR-Anweisung

Die Ausführung erfolgt nach dem folgenden Schema. Es sichert, daß der Endwert der Schleife nicht in der Schleife verändert werden kann. Außerdem kann die FOR-Schleife wie die WHILE-Schleife auch keinmal durchlaufen werden, falls nämlich der Wert von Ausdruck 1 bereits größer als der Wert von Ausdruck 2 ist:

1. Ausdruck 1 wird ausgewertet und als Startwert der Schleife gespeichert. Anschließend wird das Ergebnis von Ausdruck 2 als Endwert der Schleife gespeichert.
2. Ist der Schleifenzählerwert größer als der Endwert, wird die FOR-Anweisung beendet. Ansonsten wird er in der Laufvariablen gespeichert.
3. Die Anweisung nach dem Wortsymbol DO wird ausgeführt.
4. Der Schleifenzählerwert wird um 1 erhöht und die Ausführung der Schleife bei (2) fortgesetzt.

Das folgende kleine Programm bestimmt die Summe der Kehrwerte der Zahlen von 1 bis 100:

```
PROGRAM FORLOOP(OUTPUT);
  VAR S: REAL;
      I: INTEGER;
BEGIN
  S:= 0.0;
  FOR I:= 1 TO 100 DO
   S:= S + 1 / I;
  WRITELN(S);
END.
```

Regeln für die Laufvariable

Als Typ der Laufvariablen sind alle skalaren Typen (z.B. INTEGER, CHAR, BOOLEAN, nicht aber REAL) zulässig. Dementsprechend wird in Schritt (4) allgemein der Nachfolger im Wertebereich bestimmt:

```
PROGRAM FORCHAR (OUTPUT);
  VAR CH: CHAR;
BEGIN
  FOR CH:= ' ' TO 'Z' DO
    WRITELN('Der Code von ', CH,' ist', ORD(CH):4);
END.
```

Rückwärtszählende FOR-Schleifen

Die einzige Möglichkeit zur Änderung der Schrittweite einer FOR-Anweisung besteht darin, abwärts zu zählen:

```
FOR Variable := Ausdruck 1 DOWNTO Ausdruck 2 DO
  Anweisung
```

Die Variable durchläuft also ihren Wertebereich zwischen Ausdruck 1 und Ausdruck 2 rückwärts:

```
PROGRAM BACKANDFOR(OUTPUT);
  VAR CH: CHAR;
BEGIN
  FOR CH:= 'Z' TO 'A' DO WRITE(CH);
  WRITELN('Schleife 1 beendet');
  FOR CH:= 'Z' DOWNTO 'A' DO WRITE(CH);
  WRITELN;
  WRITELN('Schleife 2 beendet')
END.
```

Die erste Schleife wird keinmal durchlaufen, da 'Z' größer als 'A' ist, wohingegen in der zweiten Schleife die Buchstaben von 'Z' bis 'A' rückwärts gedruckt werden.

Approximative Integralberechnung

Im Beispielprogramm in Bild 23 wird das bestimmte Integral der Funktion F = SIN(X) in einem frei wählbaren Intervall mit der einfachen Sehnentrapezregel (siehe Mathematik-Schulbücher) angenähert. Dazu berechnet man (N+1) Funktionswerte an äquidistanten Stützstellen, die jeweils den Abstand H voneinander besitzen. Bildet man nun die Summe über diese Funktionswerte,

```
PROGRAM SEHTRA (INPUT,OUTPUT);
(* Berechnung eines bestimmten Integrals nach der *)
(* summierten Sehnentrapezformel:                 *)
(* (Es gibt wesentlich bessere Methoden!)         *)
  VAR SIGMA              : REAL;
      VON, BIS ,X ,F ,H  : REAL;
      I,N                : INTEGER;
BEGIN
  WRITE('Integrationsgrenzen:');
  READLN(VON, BIS);
  WRITE('Stützstellen:');
  READLN(N);
  H:= (BIS-VON) / N;
  SIGMA:= 0;
  FOR I:= 0 TO N DO
    BEGIN
      X:= VON + I * H;
      F:= SIN(X);
      IF (I=0) OR (I=N) THEN
        SIGMA:= SIGMA + F / 2
      ELSE
        SIGMA:= SIGMA + F;
    END;
  WRITELN('Näherung:', SIGMA * H);
END.
```

Bild 23: Numerische Integration

wobei die Werte am Intervallrand (I=0) oder (I=N) nur halb gewichtet werden, so ist das Produkt SIGMA * H eine Näherung für das Integral der Funktion zwischen VON und BIS.

Die Genauigkeit der Näherung hängt unter anderem von der Anzahl der Stützstellen ab. Solange nicht Rundungsfehler das Ergebnis beeinflussen, steigt die Genauigkeit mit dem Quadrat der Anzahl der Stützstellen.

Keine reellen Laufvariablen

Da als Laufvariable keine reellen Zahlen zulässig sind, ist folgende Schleife nicht in Pascal möglich:

Falsch!

```
FOR X:= VON TO BIS STEP H DO ...
```

Solche Schleifen bieten nämlich eine ideale Grundlage für undurchsichtige Fehler, falls bei der sukzessiven Addition der Schrittweite H Rundungsfehler auftreten. Als abschreckendes Beispiel sollten Sie in BASIC folgendes Programm probieren:

```
FOR I:= -1 TO 1 STEP 0.1: PRINT I: NEXT
```

Durch Rundungsfehler werden nur 20 (statt 21) Zahlen gedruckt! Die in Programm 23 gewählte Methode benötigt zwar bei jedem Schleifendurchlauf eine zusätzliche Multiplikation, jedoch können sich dort Rundungsfehler nicht kumulieren.

Geschachtelte FOR-Schleifen

Die Ausgabe einer Multiplikationstabelle ist ein anschauliches Beispiel für geschachtelte FOR-Schleifen:

```
PROGRAM MULTIPLIKATION (OUTPUT);
  CONST N=13;
  VAR I,J: INTEGER;
BEGIN
  FOR I:= 1 TO N DO
    BEGIN
      FOR J:= 1 TO N DO
        WRITE(I * J : 3);
      WRITELN;
    END;
END.
```

Dabei durchläuft die Variable J in der *inneren* Schleife die Werte 1 bis 13, während die Variable I in der *äußeren* Schleife die momentan gedruckte Zeile zählt. In jeder Zeile werden die Produkte I*J gedruckt.

2.8.7 Sprunganweisung

Syntax und Semantik der Sprunganweisung

In diesem Abschnitt werden enzyklopädisch alle Regeln zusammengestellt, die die Sprunganweisung betreffen. Deshalb werden einige Begriffe auftreten, die erst in späteren Abschnitten erklärt werden.

Sprunganweisungen sollten nur zur Behandlung selten auftretender Ausnahmefälle in einem Programm benutzt werden. Außerdem ist es sinnvoll, bei einer Sprunganweisung Kommentare einzufügen, die erläutern, wohin und warum gesprungen wird.

Eine Sprunganweisung hat die folgende Syntax:

```
GOTO Label
```

Die Programmausführung wird beim Erreichen der Sprunganweisung an der Anweisung fortgesetzt, die durch das Label markiert wird. Ein Label ist eine positive ganze Zahl. Jede Anweisung kann durch ein Label markiert werden:

```
Label : Anweisung
```

Alle Labels müssen außerdem in dem Block, in dem sich die markierte Anweisung befindet, im Labeldeklarationsteil aufgeführt werden. Im *report* wird definiert, daß der Labeldeklarationsteil die erste Deklaration eines Blockes bilden muß. In *Pascal 2.0* ist, wie bereits erwähnt, die Reihenfolge der Deklarationsteile frei wählbar. Eine Labeldeklaration hat folgende Syntax:

```
LABEL Label, Label, ... , Label;
```

GOTO für Fehlerbehandlungen

Sinnvolle Beispiele für Sprünge findet man nur in komplexen Programmen, bei denen eine Ausnahmebehandlung erforderlich wird. In kleinen Programmen läßt sich die Sprunganweisung ohne Mehraufwand problemlos vermeiden. Daher ist das folgende Programmstück nur als ein künstliches Beispiel zu betrachten (BASIC läßt grüßen):

```
PROGRAM SPRUNG(OUTPUT);
  LABEL 1,7,8;
  VAR   X: INTEGER;
BEGIN
  WRITELN('Punkt 1');
  GOTO 7;
  1: IF X=0 THEN GOTO 8;
  WRITELN('Punkt 2', X);
  X:= X-1; GOTO 1;
  7: X:=5; GOTO 1;
  8: WRITELN('Punkt 3');
END.
```

Das Programm erzeugt folgende Ausgabe:

```
Punkt 1
Punkt 2: 5
Punkt 2: 4
Punkt 2: 3
Punkt 2: 2
Punkt 2: 1
Punkt 3
```

Sprünge und die Blockstruktur

Bei Sprüngen muß die durch Prozeduren, Funktionen und zusammengesetzte Anweisungen gebildete Blockstruktur eines Pascal-Programmes beachtet werden:

1. Es ist nicht erlaubt, von außen in eine Prozedur (Funktion) zu springen.
2. Es ist nicht erlaubt, von außen in eine FOR-Anweisung, WITH-Anweisung oder CASE-Anweisung zu springen.

Andererseits ist es erlaubt, aus einer geschachtelten Prozedur zu einer Marke in einem umfassenden Block zu springen. Beispiele für diese Regeln bei blockübergreifenden Sprüngen finden Sie in Abschnitt 4.4.2 unter dem Stichwort »Sprunganweisung«.

Aufgaben

1. Schreiben Sie ein Programm zur Umwandlung arabischer Zahlen in römische Zahlen. Vielleicht haben Sie auch eine Idee, wie man die Umwandlung in umgekehrter Richtung realisieren kann.

2. Schreiben Sie ein Programm, das Zahlenfolgen einliest und bei der Eingabe der Zahl 999 die Summe über die Folge (natürlich ohne 999) druckt. Welche Wiederholungsanweisung ist hier angebracht?
3. Erweitern Sie das Programm 2, so daß die größte und kleinste Zahl der eingelesenen Zahlen bestimmt wird:
   ```
   1 3 44 33 -20 12 999
   SUMME:     73
   MAXIMUM:   44
   MINIMUM: -20
   ```
4. Erweitern Sie das Programm aus Abschnitt 2.8.5, indem Sie vor dem Zeichnen der Funktion zunächst den Bereich der auftretenden Y-Werte bestimmen. Nachdem der maximale und minimale Y-Wert bekannt sind, können Sie die Y-Koordinaten wie die X-Koordinaten skalieren, um die Funktion auch in Y-Richtung immer bildschirmfüllend darzustellen.
5. Erstellen Sie ein Programm, das die Nullstellen einer Funktion F nach dem Intervallschachtelungsverfahren bestimmt: Der Benutzer gibt als Startwerte die Intervallgrenzen L und R an, zwischen denen eine Nullstelle liegt. Dabei soll gelten:
   `F(L) * F(R) <= 0`
   d.h. zwischen L und R liegt ein Vorzeichenwechsel. Die Nullstelle wird durch sukzessive Intervallhalbierung gefunden. In Abhängigkeit vom Vorzeichen des Funktionswertes in der Intervallmitte M=(L+R)/2 wird M als rechte oder linke Intervallgrenze des folgenden Schleifendurchlaufs benutzt. Die Approximation beende man, falls die Intervallgröße kleiner als eine vom Benutzer vorgegebene Genauigkeit ist. Prüfen Sie das Programm mit der Funktion
   `F:= 2 * SIN(X) - COS (2 * X)`
   im Intervall L=0 und R=1!
6. Schreiben Sie ein Programm, das alle Primzahlen in einem vom Benutzer vorgegebenen Intervall druckt. (Eine Zahl X heißt prim, wenn sie nur durch 1 und sich selbst geteilt werden kann. 1 ist keine Primzahl.) Die *Geradeaus*-Methode zur Programmierung besteht darin, den Divisionsrest (Operation MOD !) bei der Division der Zahl X durch alle kleineren Zahlen Y zu untersuchen. Um die Laufzeit des Programmes nicht sinnlos anwachsen zu lassen, sollten Sie sich vor der Programmierung überlegen, welche Zahlen von Anfang an als Primzahlen ausscheiden und bis zu welcher Zahl Y die Teilbarkeit geprüft werden muß.
7. Schreiben Sie ein Programm, das eine *Wahrheitstabelle* in der folgenden Form druckt:

```
 A       ¦ B       ¦ A AND B
--------+---------+----------
 FALSE   ¦ FALSE   ¦ FALSE
 FALSE   ¦ TRUE    ¦ FALSE
 TRUE    ¦ FALSE   ¦ FALSE
 TRUE    ¦ TRUE    ¦ TRUE
```

Geben Sie sich ruhig etwas Mühe bei der Formatierung. Benutzen Sie (String-)Konstanten! Besonders schön wäre es, wenn Sie zwei geschachtelte Schleifen mit booleschen Variablen A und B verwenden würden:

```
FOR A:= FALSE TO TRUE DO
...
  FOR B:= FALSE TO TRUE DO
  ...
...
```

Prüfen Sie mit dem Programm die Ergebnisse von A<B, A>B, A<=B und A>=B statt A AND B!

8. Überlegen Sie sich, wie man die Operation I MOD 7 verwenden kann, um die Ausgabe von Ergebnissen so zu steuern, daß jeweils sieben Werte in einer Zeile erscheinen:

```
 2   4   6   8  10  12  14
16  18  20  22  24  26  28
30  32  34  36  38  40  42
...
```

9. Besonders schöne Bildschirmgrafiken erhält man durch die Darstellung mathematischer Kurven, die als *Zykloiden* bezeichnet werden. Sie entstehen mechanisch als Bahn eines Punktes P, der mit einem Kreis vom Radius R im Abstand A vom Mittelpunkt M des Kreises fest verbunden ist, wenn dieser Kreis um einen anderen Kreis mit dem Radius R1 rollt. Ein Spielzeug zum Zeichnen dieser Bilder heißt *Spirograph*, siehe Bild 24.
Natürlich lassen sich die Linien auch punktweise mathematisch berechnen: Dreht sich der Kreis um einen Winkel PHI, so befindet sich der Punkt an folgenden relativen Koordinaten zum Mittelpunkt des feststehenden Kreises M1:
X = (R + R1) * COS(PHI) – A * COS((R + R1) * PHI / R)
Y = (R + R1) * SIN(PHI) – A * SIN((R + R1) * PHI / R)
Programmieren Sie diese Kurven! Für die Eingabe von R, R1, A und einer Schrittweite DELTAPHI wird eine Folge von X/Y-Koordinatenpaaren berechnet, die wie in Abschnitt 2.8.5 beim Zeichnen der Funktion mit LOCATE und DRAW zu einer zusammenhängenden Kurve verbunden werden.

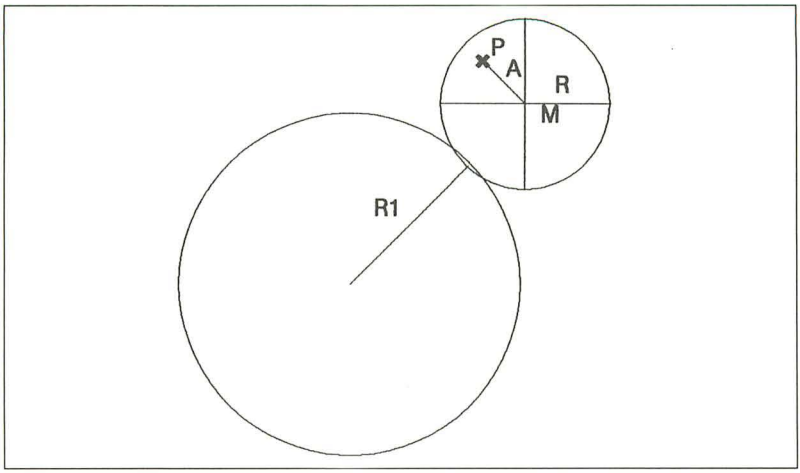


Bild 24: Parameter einer Zykloide

Da es ein nichttriviales Problem ist, aus dem rationalen Verhältnis von R und R1 den Winkel ENDPHI zu bestimmen, bei dem sich die Koordinaten zyklisch wiederholen und die Kurve sich schließt, sollten Sie die Berechnung einfach bei einer Tastenbetätigung des Betrachters beenden.

Beispielparameter:	R	=	40	20
	R1	=	35	50
	A	=	75	75
	DELTAPHI	=	0.2	1

10. Etwas aufwendig ist die Beschriftung von Grafiken: Erweitern Sie das Programm zum Zeichnen einer Funktion, indem Sie z.B. die X-Achse im Bild einzeichnen und in einem festen Abstand mit den X-Werten beschriften. Am schwierigsten ist dabei die Ausrichtung der Grafik am Text, da Text im Grafikmodus nur in 8-Pixel-Schritten ausgegeben werden kann.
Um z.B. die Zahlen von 0 bis 10 jeweils 16 Pixel in Y-Richtung auseinanderstehend zu drucken, kann man folgendes Programm verwenden:

```
PROGRAM GRAPHICTEXT(INPUT,OUTPUT,GRAPHIC);
  VAR I: INTEGER;
BEGIN
  GRAPHIC(1,1);
  FOR I:= 0 TO 10 DO
    DISPLAY(1,5,24-2*I, STR(I:5:1));
  REPEAT UNTIL KEYPRESSED;
  GRAPHIC(0);
END.
```

2.9 Die Datenstruktur Array

Arrays fassen gleichartige Objekte zusammen

In der Praxis kommt man mit den bisher besprochenen einfachen Datentypen nicht aus. Will man nämlich auf einer Menge gleichartiger Werte die gleiche Operation wiederholen, so müßte man für jedes Objekt einen eigenen Namen definieren und diese Operation mit jeder Variablen einzeln durchführen.
Eine Addition von 100 Gehältern möchte man natürlich nicht als

```
VAR GEHALT1, GEHALT2, ..., GEHALT100: REAL;
SUMME:= GEHALT1 + GEHALT2 + .... + GEHALT100
```

programmieren. Für solche Probleme vereinbart man – anschaulich gesprochen – eine Tabelle aller Werte. Nur die Tabelle erhält einen Namen, so daß jeder einzelne Wert über den Bezeichner der Tabelle zusammen mit einem **Index** in der Tabelle angesprochen wird. Einen solchen Datentyp nennt man in Pascal ein **Array** (Feldvariable).
Die einfachste Form eines Arrays besitzt als Elemente einzelne (skalare) Werte. Dabei spielen Arrays von Zeichen und Strings eine besondere Rolle. Im Abschnitt 2.9.3 werden mehrdimensionale Arrays, also Arrays von Arrays, besprochen.

2.9.1 Eindimensionale Arrays

Ein Beispiel für eindimensionale Arrays

Zunächst wird ein Beispielprogramm mit Arrays vorgestellt, damit Sie sich eine intuitive Vorstellung von den Anwendungsmöglichkeiten von Array-Variablen machen können:

```
PROGRAM GEHALTSKONTO (INPUT, OUTPUT);
  CONST N= 8;
  VAR   GEHALT: ARRAY [1..N] OF REAL;
        MAXGEHALT: REAL;
        OPERATION: CHAR;
        I        : INTEGER;
BEGIN
  FOR I:= 1 TO N DO
    GEHALT[I]:= 1000;
  REPEAT
    FOR I:= 1 TO N DO
      WRITE(GEHALT[I] : 6: 2);
    WRITELN;
    WRITELN('+  Gehaltserhöhung');
    WRITELN('-  Gehaltskürzung');
    WRITELN('=  Gehaltsangleich');
    WRITELN('*  Programmende');
    READLN(OPERATION);
```

```
    CASE OPERATION OF
     '+': BEGIN
            WRITE('Für Konto :'); READLN(I);
            GEHALT[I]:= GEHALT[I] + 50;
          END;
     '-': BEGIN
            WRITE('Für Konto :'); READLN(I);
            GEHALT[I]:= GEHALT[I] - 50;
          END;
     '=': BEGIN  (* suche maximalen Verdienst *)
            MAXGEHALT:= GEHALT[1];
            FOR I:= 2 TO N DO
              IF GEHALT[I]>MAXGEHALT THEN
                MAXGEHALT:= GEHALT[I];
            (* Nivelliere Gehälter: *)
            FOR I:= 1 TO N DO
              GEHALT[I]:= MAXGEHALT;
          END;
     '*':
     ELSE WRITE( #7);
    END; (* CASE *)
  UNTIL OPERATION = '*';
END.
```

Das Programm verwaltet eine Tabelle von N = 8 Gehältern. Aus einem Menü kann der Benutzer folgende Operationen wählen:

1. Ein einzelnes Gehalt kann gezielt erhöht werden.
2. Ein einzelnes Gehalt kann gezielt gekürzt werden.
3. Alle Gehälter können an das maximale Gehalt der Tabelle angepaßt werden.

Deklaration

Arrays werden wie alle anderen Variablen im Deklarationsteil aufgeführt. Der **Typ** Array wird folgendermaßen beschrieben:

```
ARRAY [ Indextyp ] OF Elementtyp
```

Beispiele für die Deklaration von Array-Variablen sind

```
CONST MAXINDEX=5;
VAR   A: ARRAY [5..8] OF INTEGER;
      T: ARRAY [0..MAXINDEX] OF REAL;
```

Der *Elementtyp* bezeichnet den Typ der im Array gespeicherten Werte. Bei der Gehaltstabelle und dem Array T werden reelle Zahlen gespeichert, während in dem Array A ganze Zahlen gespeichert werden. Schon jetzt können Sie sich merken, daß als *Elementtyp* **jeder** Typ der Sprache Pascal zulässig ist.

Indizierung

Um nun gezielt auf die einzelnen Elemente eines Arrays zuzugreifen, besitzt jedes Element einen **Index**. Durch den Variablennamen, gefolgt von einem

Indexausdruck in eckigen Klammern, wird ein Element des Arrays indiziert (spezifiziert):

```
WRITELN(GEHALT[1]);
GEHALT[3]:= 3000.25;
GEHALT[I]:= MAXGEHAHLT;
IF GEHALT[I]>MAXGEHALT THEN MAXGEHALT:= GEHALT[I];
```

Die Menge der zulässigen Indizes wird bereits bei der Deklaration durch den *Indextyp* festgelegt. In den Beispielen läßt sich die Variable GEHALT mit den ganzen Zahlen zwischen 0 und N indizieren, während für das Array A nur die Zahlen 5, 6, 7 und 8 erlaubt sind. Der *Indextyp* bestimmt also die Größe eines Feldes.

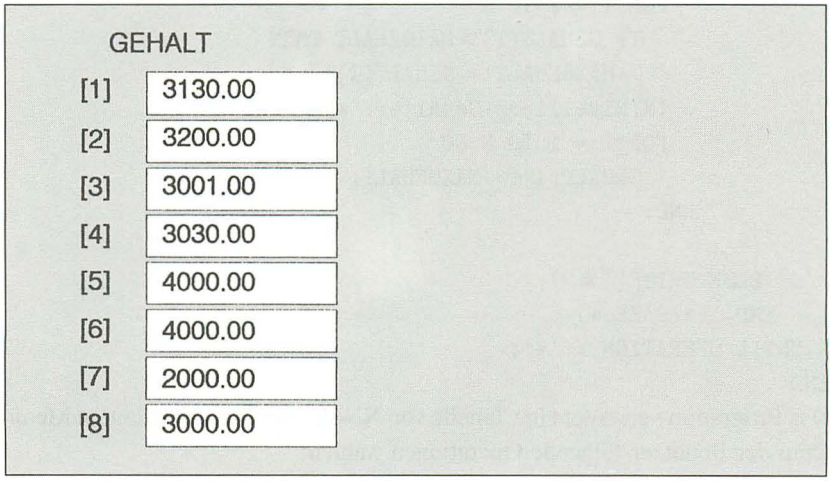


Bild 25: Struktur der Variablen GEHALT

Regeln für Indexausdrücke

Indizierte Variable können wie andere Variable in Zuweisungen, Ausdrücken und Parameterlisten verwendet werden. Jedoch muß der Typ des Indexausdruckes immer mit dem bei der Deklaration vereinbarten Indextyp übereinstimmen:

Falsch!

```
GEHALT[3.5]:= 30.0
```

Die reelle Konstante 3.5 ist also nicht als Index zulässig. Jedoch kann der Index auch durch eine Variable oder als Ergebnis eines komplexen Ausdruckes definiert werden:

```
(VAR I: INTEGER;)
GEHALT[I]:= GEHALT[(I+1) MOD N]
```

Die FOR-Anweisung wird am häufigsten im Zusammenhang mit Arrays benutzt. Man läßt die Laufvariable den Indexbereich überstreichen und kann so in einer Schleife eine Operation auf alle Elemente des Arrays anwenden:

```
PROGRAM ARRAYO (INPUT,OUTPUT);
  VAR I: INTEGER;
      A: ARRAY[-5..5] OF REAL;
BEGIN
  FOR I:= -5 TO 5 DO
    A[I]:= SQR(SIN(I));
  FOR I:= -5 TO 5 DO
    WRITELN('ELEMENT', I:3, 'ENTHAELT DEN WERT ',A[I]);
END.
```

Arrays besitzen eine konstante Größe

In Pascal müssen die Indexgrenzen bei der Deklaration durch **Konstanten** gegeben sein. Somit muß bereits bei der Übersetzung die endgültige Größe des Arrays festgelegt werden. Um diese Einschränkung etwas zu mildern, definiert man die Indexgrenzen mit Konstanten, die im Konstanten-Vereinbarungsteil zuvor definiert wurden:

```
CONST ANZAHLMITARBEITER = 2000;
VAR   GEHALT : ARRAY [1..ANZAHLMITARBEITER] OF REAL;
```

Natürlich müssen diese Konstanten dann auch im Anweisungsteil z.B. als Grenze für FOR-Anweisungen benutzt werden. Stellt sich heraus, daß die Grenze zu groß oder zu klein gewählt wurde, muß man nur die Konstante im Vereinbarungsteil anpassen und das Programm neu übersetzen.

Beispiele für Indextypen

In den bisherigen Beispielen wurde als Indextyp immer ein Teilbereich der ganzen Zahlen deklariert. In einzelnen Fällen kann es jedoch auch sinnvoll sein, andere Typen als Indizes zu vereinbaren. Außer dem Typ REAL kann man alle einfachen Typen verwenden:

```
VAR ZEICHEN: ARRAY [BOOLEAN] OF CHAR;
    ZAHLEN : ARRAY [CHAR] OF INTEGER;
    BUCHSTABEN: ARRAY ['A'..'Z'] OF INTEGER;
```

Für das letzte Beispiel zeigt Bild 26 noch ein vollständiges Programm. Es wird ein Text von der Tastatur gelesen und die Häufigkeit aller Buchstaben gezählt. Die Texteingabe wird durch die Eingabe eines beliebigen Sonderzeichens beendet:

```
    PROGRAM HAEUFIGKEIT (INPUT, OUTPUT);
    CONST SPACE = ' ';
    VAR   C: CHAR;
          H: ARRAY ['A'..'Z'] OF INTEGER; (* ZAEHLARRAY *)
  BEGIN
    (* ALLE ZAEHLER LOESCHEN *)
    FOR C:= 'A' TO 'Z' DO H[C]:= 0;
    READ(C);
```

Bild 26: Buchstaben-Statistik (Fortsetzung nächste Seite) ►

```
    WHILE (C>='A') AND (C<='Z') OR (C=SPACE) DO
      BEGIN
        IF C<>SPACE THEN H[C]:= H[C] + 1;
        READ(C);
      END;
    FOR C:= 'A' TO 'Z' DO
      WRITE(C:2, H[C]:4, '-MAL, ');
    WRITELN;
  END.
```

Bild 26: Buchstaben-Statistik (Schluß)

Typische Array-Operationen

Im Rest dieses Kapitels lernen Sie noch einige typische Operationen auf Arrays kennen. Dabei wird der folgende Deklarationsteil vorausgesetzt:

```
PROGRAM FELDER (INPUT,OUTPUT);
  CONST UG =  1;     (* untere Grenze *)
        OG = 10;     (* obere Grenze  *)
  VAR   I, J, K      : INTEGER;
        A            : ARRAY [UG..OG] OF INTEGER;
        SUM, MAX, W  : INTEGER;
```

Durchlaufen des Arrays

Wie bereits erwähnt, wird eine Operation auf allen Elementen eines Arrays mit der FOR-Anweisung programmiert:

```
SUM:= 0;
FOR I:= UG TO OG DO SUM:= SUM + A[I];
WRITELN('Die Summe aller Elemente in A ist', SUM);
```

Maximum bestimmen

Um den größten Wert in einem Array zu finden, bestimmt man schrittweise das Maximum der ersten I Zahlen, indem man den bis dahin größten Wert (der I–1 letzten Zahlen) mit dem Element A[I] vergleicht:

```
MAX:= A[UG];
FOR I:= UG+1 TO OG DO
  IF A[I]>MAX THEN MAX:= A[I];
WRITELN('Die größte Zahl ist', MAX);
```

Sortieren durch Auswahl des Maximums

Selbst wenn UG = OG ist, arbeitet dieses Programmstück korrekt, da dann die FOR-Schleife (als *pre check loop*) wegen UG+1 >OG nicht ausgeführt wird. Mit diesem Suchalgorithmus kann man auch ein Array **sortieren**: Man bestimmt das Maximum des Arrays und vertauscht es mit dem letzten Element im Array. Anschließend wiederholt man diese Prozedur für alle Elemente ohne das Maximum und erhält dadurch den zweitkleinsten Wert. Wiederholt man dieses Verfahren bis zum kleinsten Element des Arrays, so ist das Feld anschließend aufsteigend sortiert:

```
FOR J:= OG DOWNTO UG+1 DO
  BEGIN
    (* BESTIMME K, DEN INDEX DES MAXIMUMS *)
    (* IM TEILARRAY A[UG..J].             *)
    (* VERTAUSCHE A[J] MIT A[K]           *)
  END;
```

Die in Kommentarklammern angegebenen Operationen wurden bereits früher besprochen (siehe oben), so daß man diese Programmteile zwischen BEGIN und END unverändert einsetzen kann:

```
PROGRAM FELDER (INPUT,OUTPUT);
  CONST UG =  1;     (* untere Grenze *)
        OG = 10;     (* obere Grenze  *)
  VAR   I, J, K      : INTEGER;
        A            : ARRAY [UG..OG] OF INTEGER;
        MAX          : INTEGER;
BEGIN
  WRITELN('Unsortierte Folge:');
  (* A zufällig vorbelegen: *)
  FOR I:= UG TO OG DO
    BEGIN
     A[I]:= INT(RANDOM(0) * 100); WRITE(A[I]);
    END;
  WRITELN;
   FOR J:= OG DOWNTO UG+1 DO
    BEGIN
      MAX:= A[UG]; K:= UG;
      FOR I:= UG+1 TO J DO
        IF A[I]>MAX THEN
          BEGIN K:=I; MAX:= A[I]; END;
      A[K]:= A[J]; A[J]:= MAX;
    END;
  WRITELN('Sortierte Folge:');
  FOR I:= UG TO OG DO WRITE(A[I]);
  WRITELN;
END.
```

Ist Ihnen die genaue Funktion dieses Programmes noch nicht ganz klargeworden, sollten Sie mit Stift und Papier das Array mit den Werten

8 9 1 3 4 2 5

nach dem angegebenen Algorithmus sortieren. Dann werden Sie auch verstehen, warum diese Sortiermethode *Sortieren durch Auswahl* heißt. In Abschnitt 2.11.5 werden Sie ein anderes Sortierverfahren kennenlernen, das

Beispiel zum Sortieren

für große Arrays wesentlich effizienter ist. In jedem Durchlauf der äußeren Schleife (FOR J:= ...) wird eine Zahl an ihre korrekte Position gestellt:

8	5	1	3	4	2	*9*
2	5	1	3	4	*8*	*9*
2	4	1	3	*5*	*8*	*9*
2	3	1	*4*	*5*	*8*	*9*
2	1	*3*	*4*	*5*	*8*	*9*
1	*2*	*3*	*4*	*5*	*8*	*9*

Suche in Arrays

Eine weitere elementare Operation mit Arrays besteht darin, einen vorgegebenen Wert im Array zu suchen. Das Ergebnis der Suche soll der Index des Wertes im Array sein. Das Prinzip besteht darin, mit einer Variablen (I) schrittweise das Array zu durchlaufen und das Element A[I] mit dem gesuchten Wert W zu vergleichen, bis eine Übereinstimmung eintritt oder das Ende des Arrays erreicht wird. Offensichtlich ist hier die FOR-Anweisung ungeeignet, da nicht *à priori* die Grenzen für die Laufvariable bekannt sind.

Ein erster Lösungsansatz wäre folgende Schleife:

```
PROGRAM SUCHE (INPUT,OUTPUT);
(*$R+   Bereichsüberprüfung einschalten! *)
  CONST UG =  1;     (* untere Grenze *)
        OG = 10;     (* obere Grenze  *)
  VAR   I            : INTEGER;
        A            : ARRAY [UG..OG] OF INTEGER;
        W            : INTEGER;
BEGIN
  WRITELN('Zufällige Folge:');
  FOR I:= UG TO OG DO
    BEGIN
     A[I]:= INT(RANDOM(0) * 100); WRITE(A[I]);
    END;
  WRITELN;
  WRITE('Suchwert:'); READLN(W);
  I:= UG;
  WHILE (I<=OG) AND (A[I]<>W) DO I:= I+1;
  IF I>OG THEN
    WRITELN(W, ' nicht gefunden')
  ELSE
    WRITELN(W, ' hat den Index ',I);
END.
```

Ungültige Indizes

Jedoch bewirkt diese Schleife einen Zugriff auf das nicht existierende Element A[OG+1], falls der gesuchte Wert W nicht im Array A enthalten ist: Für alle

Elemente des Arrays liefert der Vergleich A[I] < > W den Wert TRUE, so daß I im letzten Durchlauf den Wert OG+1 annimmt. Anschließend wird wiederum die Bedingung der WHILE-Anweisung ausgewertet. Zwar ist bereits das Ergebnis des ersten Teilausdruckes (I < =OG) FALSE, dennoch wird in Pascal ein boolescher Ausdruck immer **vollständig** ausgewertet. Deshalb wird bei der Berechnung des zweiten Teilausdruckes (A[I] < > W) auf das nicht existierende Element A[OG+1] zugegriffen.

Indexprüfung zur Laufzeit als Operation

Dieser verbotene Zugriff wird normalerweise in *Pascal 2.0* **nicht** erkannt, da Indexausdrücke nicht auf die Grenzen der Deklaration überprüft werden. Der Kommentar (*$R+ *) schaltet jedoch eine Option des Compilers ein, die unter anderem die Indexgrenzen bei jedem Array-Zugriff prüft, so daß beim Programmablauf eine Fehlermeldung erzeugt wird:

```
VALUE OUT OF BOUNDS: 11  1  10
  ERROR AT xxxx IN SUCHE
```

Das heißt, es wurde versucht, das 11. Element anzusprechen, obwohl in der Deklaration 1 und 10 als Indexgrenzen vereinbart wurden. Näheres zur Option (*$R+ *) und zur Interpretation von Laufzeitfehlermeldungen finden Sie in Kapitel 4.

Die obige Suche muß also umformuliert werden:

```
I:= UG-1;
REPEAT
  I:= I+1
UNTIL (A[I]=W) OR (I=OG);
```

Suche mit Marken zur Effizienzsteigerung

Eine intelligentere Version der Suche vermeidet die ständige Prüfung auf das Ende des Arrays (I=OG). Sucht man nämlich in einem Feld mit 2000 Elementen, so wird 1999mal *umsonst* I=OG ausgewertet. Der Trick besteht darin, den gesuchten Wert am Ende des Arrays als **Marke** (engl. *sentinel*) zu speichern, so daß spätestens dort die Suche abbricht. Dazu muß aber das Array um ein Element am Ende erweitert werden:

```
PROGRAM SUCHE (INPUT,OUTPUT);
(*$R+   Bereichsüberprüfung einschalten! *)
  CONST UG =  1;     (* untere Grenze *)
        OG = 10;     (* obere Grenze  *)
        OG1= 11;     (* Feldgröße mit Marke am Ende *)
  VAR   I            : INTEGER;
        A            : ARRAY [UG..OG1] OF INTEGER;
        W            : INTEGER;
BEGIN
  WRITELN('Zufällige Folge:');
  FOR I:= UG TO OG DO
    BEGIN
```

```
      A[I]:= INT(RANDOM(0) * 100); WRITE(A[I])
    END;
  WRITELN;
  WRITE('Suchwert:'); READLN(W);
  I:= UG; A[OG1]:=W;                    (* Marke speichern *)
  WHILE A[I]<>W DO I:= I+1;
  IF I=OG1 THEN
    WRITELN(W, ' nicht gefunden')
  ELSE
    WRITELN(W, ' hat den Index ',I);
END.
```

Weitere Algorithmen

Mit diesem Programm endet die Behandlung der Algorithmen auf Arrays. Sicher werden Sie den einen oder anderen Hinweis für eigene Pascal-Programme verwenden können. Jeder Leser, der nicht jedesmal das Rad neu erfinden will, sei auf die Standardliteratur zum Thema Strukturierte Programmierung verwiesen (siehe Anhang E). Besonders nützlich ist das Buch [2], in dem alle Algorithmen in Form von Pascal-Quelltexten vorgestellt und ausführlich hergeleitet werden.

Zuweisung vollständiger Arrays

Bisher wurden nur einzelne Elemente eines Arrays verändert. Möchte man z.B. den Inhalt des Feldes A in ein Feld B desselben Typs übertragen, so könnte dies wie folgt geschehen:

```
FOR I:= UG TO OG DO B[I]:= A[I]
```

In Pascal geht dies aber auch eleganter (und wesentlich schneller) mit der Zuweisung B:=A. Voraussetzung hierfür ist, daß A und B denselben Typ besitzen. Wann zwei zusammengesetzte Variablen zuweisungskompatibel sind, wird im Abschnitt 2.10 genau erklärt.

```
PROGRAM ARRAYSLICES (INPUT,OUTPUT);
  CONST UG =  1;     (* untere Grenze *)
        OG = 10;     (* obere Grenze  *)
  VAR   I            : INTEGER;
        A,B          : ARRAY [UG..OG] OF INTEGER;
BEGIN
  WRITELN('Zufällige Folge:');
  FOR I:= UG TO OG DO A[I]:= INT(RANDOM(0) * 100);
  B:=A;
  FOR I:= UG TO OG DO
    WRITELN(A[I]:8, B[I]:8);
END.
```

2.9.2 Stringoperationen

ARRAY OF CHAR: Zeichenkette fester Länge

Zur Speicherung von Texten können Arrays mit dem Elementtyp CHAR verwendet werden. Der Text wird also zeichenweise gespeichert:

```
VAR A1,A2 : ARRAY[1..8] OF CHAR;
    A3    : ARRAY[1..15] OF CHAR;
```

Diese Arrays bestehen aus acht beziehungsweise fünfzehn Zeichen. Bereits im Abschnitt 2.1 wurden Stringkonstanten beschrieben. Eine Stringkonstante mit N Zeichen Länge besitzt implizit den Typ

```
ARRAY [1..N] OF CHAR;
```

Das im *report* definierte Standard-Pascal ist bei der Behandlung von Strings sehr restriktiv. Da dort Strings Arrays sind, besitzen sie eine konstante Größe. Deshalb sind in Standard-Pascal auch nur Zuweisungen zwischen Strings gleicher Länge erlaubt:

```
A1:= A2;                    (korrekt)
A1:='12345678';             (korrekt)
A3:='ALPHA          ';      (korrekt)
A1:= A3;                    (verboten)
A3:= '.....';               (verboten)
```

Auch Vergleiche zwischen Strings sind im *report* nur zwischen Strings identischer Länge erlaubt!

```
'OTTO' < 'OTTO2'            (verboten)
'ALPHA' < 'BETA'            (verboten)
```

Um auf ein einzelnes Zeichen eines Strings zuzugreifen, kann man ein ARRAY ... OF CHAR wie jedes andere Array indizieren:

```
FOR I:= 1 TO 8 DO WRITE('!', A1[I]);
```

Mit dieser FOR-Anweisung wird der String S1 zeichenweise ausgegeben:

```
!A!L!P!H!A! ! !
```

STRING: Zeichenkette variabler Länge

Wie viele Pascal-Implementierungen, bietet *Pascal 2.0* einen vordeklarierten Typ STRING und einen großen Satz an speziellen Stringprozeduren. Dies ermöglicht die Bearbeitung von Zeichenketten **variabler Länge**:

```
VAR S1,S2: STRING;
    S3:    STRING[16];
```

Der vordefinierte Typname STRING wurde bereits in Abschnitt 2.6.6 vorgestellt. Dort wurde auch erklärt, daß eine String-Variable wie S1 und S2 einen String variabler Länge speichern kann. Die Deklaration von S3 zeigt, daß hinter dem Typnamen STRING eine Maximallänge des Strings genannt werden kann. Der Compiler reserviert also für S3 so viel Speicherplatz, daß S3 maximal 16 Zeichen umfassen kann. Bei der Zuweisung

```
S3:= 'String mit Überlänge!'
```

wird zur Laufzeit des Programmes folgende Fehlermeldung ausgegeben:

```
STRING OVERFLOW ERROR
```

Definition der Maximallänge

Entfällt die explizite Angabe einer Maximallänge, so darf ein String **nicht länger als 80 Zeichen** werden. In eckigen Klammern kann eine Maximallänge von bis zu 255 Zeichen pro String vereinbart werden.

Operationen mit Strings und ARRAYs OF CHAR

Auf Variablen vom Typ String kann wie auf *normale* Arrays indiziert zugegriffen werden:

```
S1:= 'Beispiel';
FOR I:= 1 TO LENGTH(S1) DO
  WRITE('!', S1[I]);
```

bewirkt eine zeichenweise Ausgabe des Strings S1:

```
!B!e!i!s!p!i!e!l
```

Der Inhalt eines Elementes S[I] mit einem Index I größer als LENGTH(S) ist nicht definiert. Übrigens enthält in *Pascal 2.0* das Element S[0] die aktuelle Länge des Strings S in Zeichen. Es gilt also ORD(S[0]) = LENGTH(S). Jedoch ist der indizierte Zugriff auf Einzelzeichen bei Strings nur selten erforderlich, da spezielle Funktionen und Prozeduren zur Bearbeitung von Strings existieren:

```
PROGRAM STRINGS (INPUT,OUTPUT);
  VAR S1,S2: STRING;
     T     : STRING;
     I     : INTEGER;
BEGIN
  READLN(S1); READLN(S2);
  WRITELN('>',S1,'<');
  WRITELN('>',S2:80,'<');
  WRITELN('Längen:', LENGTH(S1), LENGTH(S2));
  T:= '>' + S1 + S2 + '<';
  WRITELN('konkateniert: ', T);
  DELETE(T,1,1); DELETE(T,LENGTH(T),1);
  WRITELN('gelöscht:', T);
  T:= COPY (T, 1, 5);
  WRITELN('die ersten 5 Zeichen:', T);
  FOR I:= 5 DOWNTO 1 DO
    INSERT('.',T,I);
  WRITELN('eingefügt:',T);
  I:= POS(S1,S2);
  IF I=0 THEN
    WRITELN(S1, ' ist nicht in ', S2, ' enthalten!')
  ELSE
    WRITELN('Position von S1 in S2 :', I);
END.
```

Bild 27: Beispiele für String-operationen

Jede der Stringfunktionen und -prozeduren wird ausführlich in Kapitel 4.4.4.7 behandelt. Dort wird insbesondere das Verhalten in Sonderfällen genau besprochen. In diesem Abschnitt wird nur ein Überblick über die zur Verfügung stehenden Operationen gegeben und die grundlegenden Begriffe bei Stringoperationen vorgestellt:

- Die Maximallänge einer Stringvariablen ergibt sich aus der Typdeklaration. Gültige Maximallängen liegen zwischen 1 und 255 Zeichen pro String.
- Die aktuelle Länge einer Stringvariablen bewegt sich immer im Bereich zwischen 0 und der Maximallänge der Deklaration. Bei Zuweisungen längerer Strings erfolgt in *Pascal 2.0* ein Programmabbruch mit Fehlermeldung.
- An vielen Stellen ist in *Pascal 2.0* die Angabe eines **String-Ausdruckes** möglich. Um die Programmierung mit Strings nicht unnötig unhandlich zu gestalten, nimmt der Compiler teilweise automatisch Typumwandlungen vor. Daher kann ein Stringausdruck folgendermaßen aufgebaut sein:

Automatische Typanpassung durch den Compiler

Stringkonstante	'Beispiel'
Stringvariable	S1
Variable des Typs ARRAY ... OF CHAR	A1
Wert des Typs CHAR	'X' CH
Ergebnis einer vordefinierten Stringoperation	A1 + S1
Ergebnis einer benutzerdefinierten Stringfunktion	COPY(S1,3,4)

Strings als Zwischenergebnisse

In einem Stringausdruck darf auch kein Zwischenergebnis länger als 255 Zeichen werden. In diesem Fall wird ein »STRING OVERFLOW ERROR« gemeldet.

In *Pascal 2.0* können alle Stringoperationen beliebig geschachtelt und auch rekursiv beliebig oft aufgerufen werden. Insbesondere können die Funktionen VAL und STR auch innerhalb von WRITE- und READ-Operationen verwendet werden. Diese Eigenschaften werden deshalb so betont, da viele Implementierungen (auch *Turbo-Pascal*) einen festen Arbeitsbereich für Stringoperationen besitzen, so daß bei den obengenannten Fällen für den Benutzer unerwartete Resultate auftreten können.

Operationen mit Strings

Nachdem nun die Terminologie geklärt ist, können die im Beispielprogramm (Bild 27) verwendeten Operationen kurz angesprochen werden. Sie sollten diese Operationen mit verschiedenen Eingaben S1 und S2 testen und bei Bedarf auch eigene kleine Testprogramme schreiben:

Beim Lesen eines Strings mit **READ** werden so lange Zeichen von der Tastatur gelesen und im String gespeichert, bis der Benutzer die Return-Taste betätigt. Bei der Ausgabe mit **WRITE** gelten die in Abschnitt 2.5.1 aufgeführten Regeln zur Formatierung.

Die Funktion **LENGTH** liefert zu einem beliebigen Stringausdruck die aktuelle Länge. Das Ergebnis ist eine Zahl vom Typ INTEGER im Bereich 0 bis 255. Eine häufige Operation im Zusammenhang mit Strings ist das Verketten (Konkatenieren) einer Folge von Stringausdrücken. Diese Operation kann einerseits mit dem Pluszeichen (»+«), andererseits aber auch mit der vordefinierten Funktion CONCAT durchgeführt werden. Die Funktion CONCAT existiert in *Pascal 2.0* nur, um Programme unverändert übernehmen zu können, die mit anderen Pascal-Compilern entwickelt wurden:

```
T:= '>' + 'Anna' + 'Otto' + '<';          entspricht
T:= CONCAT ('>', 'Anna', 'Otto','<');     entspricht
T:= '>AnnaOtto<'
```

Die Länge des Ergebnisstrings ist also gleich der Summe der Teilstrings.

```
T:='Dies ist ein Test';
DELETE(T,6,3);
(* --> T = 'Dies  ein Test' *)
```

Mit diesem Aufruf der Prozedur **DELETE** werden aus der Stringvariablen T ab dem 5. Zeichen 3 Zeichen entfernt. Vielseitiger ist die Funktion **COPY**, da sie auch auf Stringausdrücke und nicht nur auf Stringvariablen angewendet werden kann:

```
WRITELN( COPY ('Dies ist ein Test', 6, 7))
```

In diesem Beispiel wird aus dem String ab dem 6. Zeichen ein Teilstring der Länge 7 selektiert ('ist ein'). Das Gegenstück zur Prozedur DELETE ist die Prozedur INSERT. Mit ihr kann man einen beliebigen Stringausdruck in eine Stringvariable ab einer frei wählbaren Position einfügen:

```
S:= 'Hände hoch, Überfall!';
INSERT ('Bank-', S, 13);
(* --> S = 'Hände hoch, Bank-Überfall!' *)
```

Suche in Strings

Die Funktion **POS** bestimmt die Position eines Stringausdruckes in einem anderen Stringausdruck:

```
WRITELN( POS ('Wirth', 'Niklaus Wirth'));
(* Ergebnis ist die Zahl 9 *)
```

Ist der erste String nicht im zweiten enthalten, so ist das Funktionsergebnis die Zahl 0. Ebenfalls im *report* nicht vorhanden sind die folgenden Funktionen, die Umwandlungen zwischen Zahlen und Zeichenketten vornehmen.

```
PROGRAM KONVERTIERUNG (INPUT,OUTPUT);
  VAR S: STRING;
      R: REAL;
BEGIN
  R:= RANDOM(0);
  S:= STR(R) + '!' + STR (R : 14 : 8);
  WRITELN(S);
```

```
  R:= VAL(S);
  WRITELN(R + 2.0);
END.
```

Die Funktion **STR** wandelt ihr Argument (eine ganze oder reelle Zahl) gemäß den Formatierungsregeln für reelle Zahlen bei der Prozedur **WRITE** in eine Zeichenkette um. Deshalb sind sowohl Exponential- als auch Festkommadarstellung möglich. Die umgekehrte Umwandlung ist mit der Funktion **VAL** möglich: Das Argument ist ein Stringausdruck, der eine Ziffernfolge enthalten muß, die der Syntax für reelle Zahlen entspricht. Der Wert dieser Ziffernfolge wird als Funktionsergebnis übergeben:

Zahlen und Strings

```
IF VAL ('2.0') = VAL ('000002.0000') THEN
  WRITELN('Beide Zahlen sind gleich');
```

Natürlich können Sie diese Menge an Informationen nicht durch einmaliges Durchlesen verarbeiten. Jedoch gibt es sehr viele Anwendungen für Stringoperationen, bei denen sehr unterschiedliche Operationen benötigt werden. Schließlich besteht die *Kunst des Programmierens* auch in der Benutzung des richtigen Befehls zur rechten Zeit.

2.9.3 Mehrdimensionale Arrays

Arrays aus Arrays

Der Elementtyp eines Arrays ist in Pascal beliebig. Deshalb kann insbesondere auch ein Array aus Arrays gebildet werden:

```
PROGRAM MATRIX(INPUT, OUTPUT);
  CONST N=3; M=4;
  VAR   X: ARRAY [1..N] OF
              ARRAY [1..M] OF INTEGER;
        I,J : INTEGER;
BEGIN
  FOR I:= 1 TO N DO
    FOR J:= 1 TO M DO
       X[I,J]:= I * 4 + J - 4;
  FOR I:= 1 TO N DO
    BEGIN
      FOR J:= 1 TO M DO WRITE(X[I,J]:5);
      WRITELN
    END;
END.
```

Bild 28: Programm Matrix

Eine solche zweidimensionale Datenstruktur bezeichnet man in der Mathematik als eine **Matrix**. Man kann sich die Variable X als eine zweidimensionale Tabelle mit ganzen Zahlen vorstellen:

X	[1]	[2]	[3]	[4]
[1]	1	2	3	4
[2]	5	6	7	8
[3]	9	10	11	12

Bild 29: Matrix X

Deklaration

Üblicherweise wird die obige Deklaration eines mehrdimensionalen Arrays folgendermaßen abgekürzt:

```
VAR X: ARRAY[1..N, 1..M] OF INTEGER;
```

Doppelte Indizierung

Wie Sie im Beispielprogramm sehen, spricht man Elemente einer solchen Struktur durch zwei Indizes an:

```
X[1,2]    := X[2,1]
```

oder ausführlich

```
X[1] [2] := X[2] [1]
```

In der ausführlichen Schreibweise (die in der Praxis nur sehr selten anzutreffen ist) wird die schrittweise Indizierung besonders deutlich: Zunächst wird eines der Arrays 1 bis N gewählt. In dem so spezifizierten Array erfolgt dann wieder ein indizierter Zugriff auf eines der Elemente von 1 bis M. Dieses Element enthält schließlich einen Wert vom Typ INTEGER.

Es ist eine reine Konvention, bei solchen Matrizen den ersten Index als Zeile und den zweiten Index als Spalte zu bezeichnen. Diese Namensgebung orientiert sich nur an der in der Mathematik üblichen Darstellung von Matrizen, wie sie auch in Bild 29 verwendet wurde.

Formatierte Ein- und Ausgabe

Am Beispiel der Ein- und Ausgabe von Matrizen können Sie Ihre Kenntnisse über die Standardprozeduren READ und WRITE wieder auffrischen. In Bild 28 wurde die Matrix zeilen- und spaltenweise ausgegeben. Diese Formatierung wurde erreicht, indem nach dem Ende der Ausgabe jeder Zeile (*innere* FOR-Anweisung) mit WRITELN ein Zeilenvorschub erzeugt wurde.

Möchte man eine Matrix über die Tastatur zeilenweise einlesen, so kann man analog die Prozeduren READ und READLN verwenden. Gibt der Benutzer die Elemente in folgendem Format ein,

```
1   2   3   4
5   6   7   8
9  10  11  12
```

soll gelten: X[1,2] = 2 und X[3,1] = 9. Daher benutzt man wiederum zwei geschachtelte FOR-Anweisungen. Eine Laufvariable (i) indiziert die Zeilen, die andere (j) durchläuft in jeder Zeile die Spalten.

```
FOR I:= 1 TO N DO
  BEGIN
    FOR J:= 1 TO M DO
      READ( X[I,J]);
    READLN;                    (*ignoriere den Rest der Zeile *)
  END
```

Zeilen und Spalten vertauscht

Die Prozedur READLN sorgt also dafür, daß die Eingabe korrekt in verschiedenen Zeilen erfolgt. Um Mißverständnissen vorzubeugen, zeigt das nächste Programm, wie die gleiche Matrix auch *transponiert* gedruckt werden kann. Das bedeutet, daß in jeder Ausgabezeile Elemente mit dem gleichen zweiten Index erscheinen. Es genügt also, die *innere* mit der *äußeren* FOR-Anweisung gegenüber Bild 28 zu vertauschen:

```
FOR J:= 1 TO M DO
  BEGIN
    FOR I:= 1 TO N DO
      WRITE( X[I,J]:5);
    WRITELN;
  END
```

Ausgabe:

```
1    5    9
2    6   10
3    7   11
4    8   12
```

Matrizen in der Mathematik

In der Mathematik existieren zahllose Anwendungen für Matrizen mit reellen oder ganzzahligen Koeffizienten. Die mathematischen Algorithmen besitzen meist sehr übersichtliche Formulierungen in Pascal, jedoch ist im Rahmen dieses Buches eine Besprechung des theoretischen Hintergrundes nicht möglich. Auf der beiliegenden Diskette findet sich ein Programm (NEWTON.P), das eine Matrix durch ein Iterationsverfahren *invertiert*. Das folgende Programm verwendet ebenfalls Verfahren aus der Mathematik, jedoch sind die zugrundeliegenden Überlegungen etwas einfacher darzustellen.

```
PROGRAM FAHRPLAN (INPUT, OUTPUT);
  CONST STATIONEN = 4;
  VAR   VERBINDUNG, VERBINDUNGO, VERBINDUNGNEU
             : ARRAY [1..STATIONEN, 1..STATIONEN]
                 OF BOOLEAN;
        VON,NACH,UEBER: INTEGER;
        UMSTEIGEN     : INTEGER;
        WEG           : BOOLEAN;
        CH            : CHAR;
BEGIN
  WRITELN('GEBEN SIE DIE VERBINDUNGEN ZWISCHEN DEN');
  WRITELN(STATIONEN,'*',STATIONEN,' STATIONEN IN');
  WRITELN('FOLGENDEM FORMAT EIN:');
  WRITELN('     + - + -');
  WRITELN('     + + + -');
  WRITELN('     - - - +');
  WRITELN('     - + + +');
  (* DIREKTE VERBINDUNGEN EINLESEN: *)
  FOR VON:= 1 TO STATIONEN DO
   BEGIN
      FOR NACH:= 1 TO STATIONEN DO
       BEGIN
          REPEAT (* '+' oder '-' lesen *)
            READ(CH);
         UNTIL (CH ='+') OR (CH='-');
         VERBINDUNG[VON, NACH]:= CH = '+';
       END;
     READLN;
    END;
  VERBINDUNG:= VERBINDUNGO;
  FOR UMSTEIGEN:= 0 TO STATIONEN DO
    BEGIN
     WRITELN(' VERBINDUNGEN MIT 0-', UMSTEIGEN,
             '-MAL UMSTEIGEN:');
     (* ZEIGE VERBINDUNGEN FORMATIERT AN: *)
     FOR VON:= 1 TO STATIONEN DO
       BEGIN
        FOR NACH:= 1 TO STATIONEN DO
          IF VERBINDUNG[VON,NACH] THEN
```

Bild 30: Verbindungen bestimmen

```
            WRITE('+':3)
          ELSE
            WRITE('-':3);
        WRITELN;
        END;
      WRITELN;
      (* NEUE VERBINDUNGEN BERECHNEN: *)
      FOR VON:= 1 TO STATIONEN DO
        FOR NACH:= 1 TO STATIONEN DO
          BEGIN
            WEG:= FALSE;
            FOR UEBER:= 1 TO STATIONEN DO
              WEG:= WEG OR
                 VERBINDUNG[VON,UEBER] AND
                 VERBINDUNGO[UEBER,NACH];
            VERBINDUNGNEU[VON,NACH]:= WEG;
          END;
      VERBINDUNG:= VERBINDUNGNEU;
    END; (* FOR UMSTEIGEN *)
  END.
```

Bild 30: Verbindungen bestimmen (Schluß)

Beispiel »Verbindungen berechnen«

Das Programm löst folgendes Problem: Gegeben sind vier Stationen (z.B. Bahnhöfe in Frankfurt, München, Hamburg und Berlin). Zwischen manchen Stationen existieren direkte (Zug-) Verbindungen, zwischen anderen jedoch nicht. Außerdem ist es möglich, daß zwar eine Direktverbindung von Frankfurt nach München existiert, jedoch nicht umgekehrt.
Mit der Kenntnis der Direktverbindungen soll das Programm nun die Verbindungen berechnen, die sich durch Umsteigen ergeben: Angenommen, es existiere eine Verbindung von Hamburg nach Frankfurt und von dort eine Verbindung nach München. Mit einmaligem Umsteigen in Frankfurt kann man dann von Hamburg aus München erreichen, auch wenn zuvor keine Verbindung existierte. Läßt man mehrmaliges Umsteigen zu, so entstehen eventuell weitere Verbindungen, die das Programm ebenfalls anzeigen soll.

Fahrplanmatrix

Der erste Schritt zur Lösung einer Aufgabe in Pascal besteht meist in der Bestimmung der Datenstruktur zur Speicherung der Informationen. Wie in diesem Abschnitt nicht anders zu erwarten ist, handelt es sich hierbei um eine **Matrix** (in der Mathematik unter dem Gattungsnamen *Adjazenzmatrix* bekannt). Die Verbindungsmöglichkeiten lassen sich z.B. tabellarisch folgendermaßen darstellen:

VON/NACH	Frankfurt	München	Hamburg	Berlin
Frankfurt	ja	ja	ja	nein
München	nein	ja	nein	nein
Hamburg	ja	nein	ja	ja
Berlin	ja	nein	ja	ja

Deshalb wurde die Variable VERBINDUNG folgendermaßen deklariert:

```
VERBINDUNG: ARRAY [1..STATIONEN, 1..STATIONEN] OF BOOLEAN;
```

Numeriert man die Stationen in der obigen Reihenfolge von 1 bis 4 durch, so gilt also:

```
VERBINDUNG [1,1] = TRUE
VERBINDUNG [1,2] = TRUE
VERBINDUNG [1,3] = TRUE
VERBINDUNG [1,4] = FALSE
```

Codierung mit Typ BOOLEAN

Generell gilt VERBINDUNG[VON, NACH] = TRUE genau dann, wenn eine Verbindung von Station VON nach Station BIS existiert. Zur einfachen Ein- und Ausgabe wird eine existierende Verbindung mit »+« markiert, während eine fehlende Verbindung mit »–« markiert wird. Damit läßt sich obige Tabelle kompakt schreiben als:

```
+  +  +  -
-  +  -  -
+  -  +  +
+  -  +  +
```

Die Ein- und Ausgabe erfolgt wieder zeilenweise nach dem bereits bekannten Schema durch zwei geschachtelte FOR-Anweisungen. Offensichtlich wird die Lesbarkeit eines Programmes stark durch die Verwendung aussagekräftiger Variablennamen (VON, BIS, VERBINDUNG) erhöht.

Programmgrobstruktur

Bevor Sie sich zu sehr mit den Details des Programmes belasten, sollten Sie folgende Struktur des Algorithmus erkennen:

Codierte Matrix der Verbindungen einlesen
Für 0 bis 4 Umsteigemöglichkeiten
Zeige Matrix der Verbindungen codiert an
Berechne neue Verbindungsmatrix unter Berücksichtigung der durch ein zusätzliches Umsteigen entstehenden Verbindungen.
Übernehme die neue Matrix als Arbeitsmatrix

Wie bestimmt man aber aus dem Inhalt der Matrix VERBINDUNG die Matrix VERBINDUNGNEU? Dazu bestimmt man für **jedes** Paar von Ausgangs- und Zielstation (VON und BIS) jeden Weg, der über eine **beliebige** Zwischen-

station UEBER führt: Für VON = Hamburg und BIS = München erhält man z.B. folgende Regel:

Existenz eines Weges mit Umsteigen

Es existiert genau dann ein Weg (evtl. mit einmaligem Umsteigen) von Hamburg nach München, wenn eines der folgenden Verbindungspaare existiert:

von Hamburg über Frankfurt	nach München	ODER
von Hamburg über München	nach München	ODER
von Hamburg über Hamburg	nach München	ODER
von Hamburg über Berlin	nach München	

Diese Berechnungsregel wird durch die *innerste* FOR-Anweisung realisiert:

```
WEG:= FALSE;
FOR UEBER:= 1 TO STATIONEN DO
   WEG:= WEG OR
         VERBINDUNG[VON,UEBER] AND VERBINDUNGO[UEBER,NACH];
VERBINDUNGNEU[VON,NACH]:= WEG;
```

Indem man diese Regel nacheinander für alle Paare VON,BIS wiederholt, erhält man eine neue Matrix VERBINDUNGNEU. Wiederholt man diese Berechnung (Schleife FOR UMSTEIGEN:= ...), so erhält man auch Verbindungen mit mehrmaligem Umsteigen.

In einem weniger ausgebauten Schienennetz könnte man also folgende Folge von Matrizen erhalten:

Beispielfahrplan

```
Verbindungen mit 0-0mal Umsteigen:
+   +   -   -
-   +   +   -
-   -   +   +
+   -   -   +
Verbindungen mit 0-1mal Umsteigen:
+   +   +   -
-   +   +   +
-   -   +   +
+   +   -   +
Verbindungen mit 0-2mal Umsteigen:
+   +   +   +
+   +   +   +
+   +   +   +
+   +   +   +
```

Schnelle Blockanweisungen zwischen zusammengesetzten Typen

In dem Programm aus Bild 30 findet sich auch die Zuweisung VERBINDUNG:= VERBINDUNGNEU. Mit ihr werden alle 16 Elemente der Matrix VERBINDUNGNEU in die Matrix VERBINDUNG kopiert. Solche *Blockzuweisungen* sind in Pascal bei jedem *zusammengesetzten Typ* möglich. Bei mehrdimensionalen Arrays können dabei hierarchisch Teile der Struktur

kopiert werden. Man gibt dazu beim Variablennamen jeweils die Indizes nur bis zu der Dimension an, die komplett verändert werden soll:

```
VAR H1, H2: ARRAY [0..20] OF
                ARRAY [1..4] OF
                 ARRAY [1..10] OF INTEGER;
```

Ein Hochhaus als dreidimensionale Datenstruktur

Mit etwas Phantasie kann man in dieser Datenstruktur zwei Hochhäuser mit 20 Stockwerken (einschließlich Erdgeschoß) erkennen. In jedem Stockwerk gibt es vier Flure. Jeder Flur besitzt die Zimmernummern 1 bis 10. Für jedes Zimmer wird schließlich die Anzahl der Personen (vom Typ INTEGER) gespeichert. H1 und H2 sind also *dreidimensional*. Mit folgenden Zuweisungen kann man einige *Umzüge* durchführen:

```
H1[1,2,2]:= H1[1,2,3];  H1[1,2,3]:= 0
```

Hier zieht also eine Familie (?) aus dem 1. Stock 2. Flur in die Nachbarwohnung um. Die verlassene Wohnung bleibt leer. Wenn alle Bewohner des 3. Flurs im 2. Stock die Wohnungen mit den gleichen Zimmernummern im 4. Flur im Erdgeschoß übernehmen sollen, wird das in der Zimmerbuchführung so notiert:

```
H1[0,4]:= H1[2,3]                entspricht der Schleife
FOR ZIMMER:= 1 TO 10 DO
  H1[0,4,ZIMMER]:= H1[2,3,ZIMMER]
```

Mehrstufige Blockanweisungen

Beim Umzug des kompletten 1. Stockwerks in das 2. Stockwerk genügt folgende Zuweisung:

```
H1[2]:= H1[1]            entspricht der Schleife
FOR FLUR:= 1 TO 4 DO
  FOR ZIMMER:= 1 TO 10 DO
    H1[2,FLUR,ZIMMER]:= H1[1,FLUR,ZIMMER]
```

Größere Unruhe in beiden Hochhäusern wird wohl die folgende Aktion zur Folge haben:

```
H1:= H2                  entspricht der Schleife
FOR STOCK := 0 TO 20 DO
  FOR FLUR := 1 TO 4 DO
    FOR ZIMMER:= 1 TO 10 DO
      H1[STOCK,FLUR,ZIMMER]:= H2[STOCK,FLUR,ZIMMER]
```

Aufgaben

Binäres Suchen

1. Schreiben Sie ein Programm, das in einem aufsteigend **sortierten** Array A ganzer Zahlen nach einer vorgegebenen Zahl W sucht. Dabei soll die Methode der *binären Suche* verwendet werden: Um im Teilarray von L bis R das Element W zu finden, vergleicht man W mit dem Wert an der mittleren Position M=(L+R) DIV 2. Je nachdem, ob W kleiner oder größer als A[M] ist, wählt man M als neue linke oder rechte Arraygrenze.

```
PROGRAM BINARYSEARCH (INPUT,OUTPUT);
  CONST UG = 0; OG = 40;
  VAR   A: ARRAY [UG..OG] OF INTEGER;
        W: INTEGER;
        M,L,R: INTEGER;
BEGIN
  (* Array A zufällig sortiert (!) belegen: *)
  W:= 0;
  FOR M:= UG TO OG DO
    BEGIN
      W:= W + INT(RANDOM(0) * 12);
      A[M]:= W; WRITE(W:4);
    END;
  WRITELN;
  WRITE('Suche nach:'); READLN(W);
  L:= UG; R:= OG;
  REPEAT
    M:= (L+R) DIV 2;
    IF W<A[M] THEN ... ELSE
    IF W>A[M] THEN ...
    ELSE ...
  UNTIL ...
  IF ... THEN
    WRITELN('gefunden!')
  ELSE
    WRITELN('nicht gefunden!');
END.
```

Sie können bei der Formulierung des Abbruchkriteriums eine boolesche Variable GEFUNDEN verwenden. Wenn Sie Programmier-Erfahrung besitzen, versuchen Sie, in der REPEAT-Schleife mit nur einem Vergleich zwischen W und A[M] auszukommen. (Es geht!).

2. Schreiben Sie ein Programm, das das *Pascalsche Dreieck* druckt. Es enthält die Binomialkoeffizienten n über k, die man z.B. zur Berechnung von $(a + b)^n$ benötigt:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
... ... ... ... ... ... ...
```

Dabei steht in der n. Zeile an der k. Position die Summe der Zahlen in der n–1. Zeile an der k–1. und k. Position. Genaueres steht in jedem Lexikon.

3. Schreiben Sie ein Programm, das ein Array reeller Zahlen (z.B. Funktionswerte) verwaltet. Zu Anfang ist das Array unbelegt. Dann sollen Werte vom Benutzer eingegeben werden, die direkt bei der Eingabe so in das Array eingefügt werden, daß die Zahlen aufsteigend **sortiert** bleiben.
   Das Programm hat also folgende Grobstruktur:

```
LEN:= 0;
REPEAT
  Zahl einlesen;
  Zahl in Array an korrekter Position einfügen;
  LEN:= LEN+1
UNTIL LEN = Feldgröße
```

Blocksatz mit Strings

4. Schreiben Sie ein Programm, das einen Text zeilenweise von der Tastatur einliest. Dieser Text soll dann im Blocksatz auf eine vom Benutzer wählbare Spaltenbreite von N Zeichen verteilt ausgedruckt werden.

```
Dies    ist  ein
Beispiel für den
Blocksatz    mit
wenigen Spalten.
```

   Sie müssen die Wörter über die Zeilen verteilen und sich dazu an den Wortzwischenräumen (' ') orientieren. Anschließend ist jede Zeile auf die geforderte Länge von N Zeichen aufzufüllen, indem zusätzliche Leerzeichen gleichmäßig zwischen den Wörtern eingefügt werden.
   Verwenden Sie zur Speicherung des Textes ein Array von Strings. Sicher können Sie die Funktionen LENGTH, POS, INSERT in diesem Zusammenhang gut verwenden.

5. Schreiben Sie ein Programm, das eine reelle Zahl X gemäß der deutschen Konventionen mit Komma und Trennzeichen druckt. Dabei sollen insgesamt N Zeichen gedruckt werden, und M soll die Anzahl der Nachkommastellen definieren.

```
X:= 12.34    N=10  M=2    _____12,34
X:=1E+7      N=20  M=2    _______10.000.000,00
X:=-0.07     N=10  M=4    ___-0,0700
X:=9.876     N=10  M=0    _________9
```

   Wie bei Aufgabe 4 können Sie sich Ihre Aufgabe durch die geschickte Verwendung der vordefinierten Stringprozeduren (STR, POS, INSERT, CONCAT) stark vereinfachen.

String-vergleich mit Joker

6. Erstellen Sie ein Program, das einen String S1 in einem String S2 sucht. Die gefundene Position soll in der Variablen M gespeichert werden. Im Gegensatz zur Funktion POS darf S1 auch sogenannte *Joker* enthalten:

```
S2:= 'Dies ist ein Testtext';
S1:= 'tt'                    --> M= 17
S1:= 'Tes??e'                --> M= 14
S1:= 'ein?T'                 --> M= 10
S1:= 'Egon'                  --> M= 0
```

7. Für eine vom Benutzer eingegebene Matrix mit drei Zeilen und vier Spalten wie z.B.

   10 20 30 40
   40 30 20 10
   11 12 13 -5

   sollen folgende Werte berechnet werden:
   - Maximale Zeilensumme (100 in Zeile 1)
   - Maximale Spaltensumme (63 in Spalte 3)
   - Das betragsgrößte Element der Matrix (40)
   - Die Summe der Werte in jeder Diagonalen von links unten nach rechts oben (10, 60, 71, 72, 23, –5)
   - Die Summe der Werte in jeder Diagonalen von links oben nach rechts unten (11, 52, 53, 35, 40, 40)

   Natürlich werden Sie dafür FOR-Anweisungen verwenden und die Größe der Matrix durch Konstanten (N=3, M=4), die Sie auch bei den FOR-Anweisungen benutzen, veränderbar gestalten!

8. Programmieren Sie das Sortierverfahren **Bubblesort**. Um ein Array von N Werten zu sortieren, wird das Array (N–1)mal durchlaufen. In jedem Durchlauf werden jeweils zwei benachbarte Elemente verglichen und ausgetauscht, falls das zweite Element kleiner als das erste Element ist. **Bubblesort**

```
PROGRAM BUBBLESORT (INPUT,OUTPUT);
  CONST N = 8;
  VAR A: ARRAY[1..N] OF INTEGER;
      I,J,H: INTEGER;
BEGIN
  FOR I:= 1 TO N DO A[I]:= INT(RANDOM(0)*100);
  FOR I:= 2 TO N DO
    FOR J:= ............. DO
      IF A[J+1] < A[J] THEN
        BEGIN (* vertauschen *)
          H:= A[J]; A[J]:= A[J+1]; A[J+1]:=H;
        END;
  WRITELN('Sortierte Folge:');
  FOR I:= 1 TO N DO WRITE(A[I]:4);
  WRITELN;
END.
```

Bestimmen Sie die Grenzen für die *innere* FOR-Anweisung, indem sich zunächst die Funktionsweise mit der Anweisung FOR J:= 1 TO N-1 klarmachen, und anschließend überlegen, bis zu welcher Grenze das Array im I-ten Durchlauf bereits sortiert ist.

9. Schreiben Sie ein Programm, das eine *Rosette* wie im folgenden Bild zeichnet:

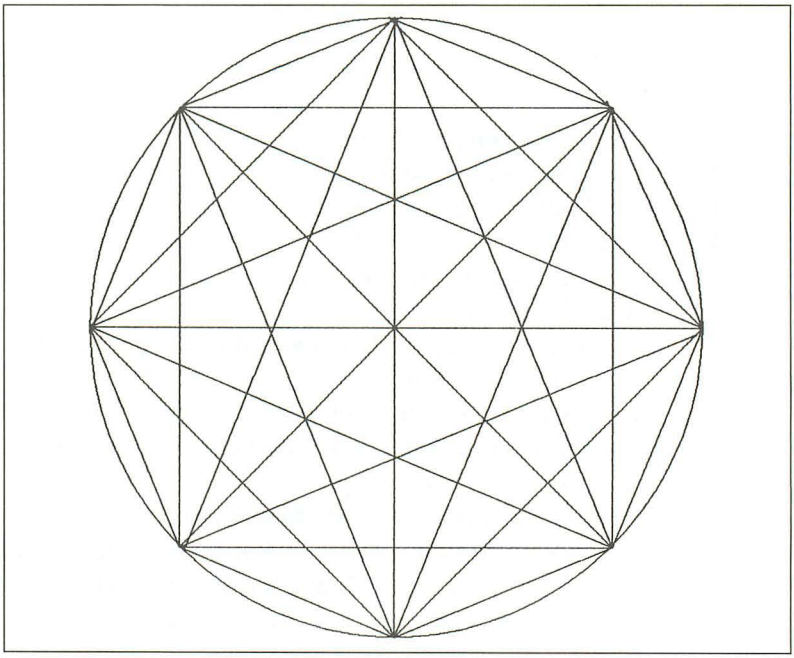


Bild 31: Eine Rosette

Der Benutzer wählt die Anzahl der Endpunkte N (im Beispiel N=8), die in gleichem Winkelabstand auf einem Kreis liegen sollen (Winkel 360 Grad $= 2 * \pi$). Nachdem Sie die Bildschirmkoordinaten dieser N Punkte berechnet haben, verbinden Sie jeden der Punkte mit allen übrigen Punkten. Vermeiden Sie, daß die Linien doppelt gezeichnet oder die Endpunkte mehr als einmal berechnet werden!

2.10 Deklaration von Typen

Benutzerdefinierte Typen

In der Variablendeklaration wird der Typ als eine konstante Eigenschaft der Variablen festgelegt. Bisher kennen Sie die **Standardtypen** mit den vordefinierten Namen INTEGER, REAL, CHAR, BOOLEAN und STRING. Im vorausgegangenen Abschnitt 2.9 wurden bereits Array-Variablen definiert. Sie sind ein erstes Beispiel für Typen, die der Programmierer selbst definiert. Wie Konstanten und Variablen können auch Typen **Namen** erhalten. Dies geschieht

im Typvereinbarungsteil, der im Programmtext zwischen dem Konstanten- und dem Variablenvereinbarungsteil steht und mit dem Wortsymbol TYPE eingeleitet wird:

Deklaration im Vereinbarungsteil

```
PROGRAM TYPES (INPUT,OUTPUT);
  CONST N = 4;
  TYPE  GANZEZAHL   = INTEGER;
        GROSSEZAHL  = REAL;
        TMATRIX     = ARRAY [1..N, 1..N] OF GANZEZAHL;
        LONGSTRING  = STRING[255];
        SHORTSTRING= STRING[16];
  VAR   I           : GANZEZAHL;
        J           : INTEGER;
        STR1        : LONGSTRING;
        STR2        : SHORTSTRING;
        MAT1, MAT2  : TMATRIX;
        MAT3        : ARRAY [1..N, 1..N] OF GANZEZAHL;
        MAT4        : TMATRIX;
BEGIN
END.
```

Bild 32: Typdeklarationen

In *Pascal 2.0* können mehrere Typdeklarationsteile existieren, die auch mit Variablendeklarationsteilen gemischt werden können. Jedoch muß jeder Typname vor seiner Anwendung deklariert werden:

```
PROGRAM TYPES (INPUT,OUTPUT);
 TYPE   GANZEZAHL   = INTEGER;
        GROSSEZAHL  = REAL;
  VAR   I           : GANZEZAHL;
        J           : INTEGER;
  CONST N = 4;
  TYPE  TMATRIX     = ARRAY [1..N, 1..N] OF GANZEZAHL;
  VAR   MAT1, MAT2  : TMATRIX;
        MAT3        : ARRAY [1..N, 1..N] OF GANZEZAHL;
        MAT4        : TMATRIX;
  TYPE  LONGSTRING  = STRING[255];
        SHORTSTRING= STRING[16];
  VAR   STR1        : LONGSTRING;
        STR2        : SHORTSTRING;
BEGIN
END.
```

Bild 33: Gemischte Deklarationsteile in Pascal 2.0

Übersicht durch Typnamen

Für zusammengesetzte Typen ist die Benutzung von Typbezeichnern besonders sinnvoll. Im letzten Abschnitt wäre die Struktur der Variablen H1 und H2 mit der folgenden Typdeklaration sofort erkennbar geworden:

```
TYPE BELEGUNG        = INTEGER;
     FLUR            = ARRAY [1..10] OF BELEGUNG;
     STOCK           = ARRAY [1..4]  OF FLUR;
     HAUS            = ARRAY [0..20] OF STOCK;
VAR  H1,H2           : HAUS;
```

Typkompatibilitätstests bereits bei der Übersetzung

Eine wichtige Aufgabe der Typdeklaration besteht darin, die Menge der Variablen in einem Programm in *Klassen* aufzuteilen. Zuweisungen und Operationen sind nur zwischen den Mitgliedern *kompatibler* Klassen möglich. Bereits bei der Übersetzung sind dadurch weitgehende Prüfungen des Programmtextes möglich.

Definition der Typkompatibilität in Pascal

Die folgenden Regeln legen die Typkompatibilität in Pascal fest. Dabei werden einige Regeln schon im Vorgriff auf spätere Kapitel aufgeführt:

1. Zwei Variablen eines zusamengesetzten Typs (Array, Record, Menge) sind nur dann zuweisungskompatibel, wenn sie in einer gemeinsamen Variablendeklaration oder mit demselben Typnamen definiert wurden.
2. Stringausdrücke können Stringvariablen zugewiesen werden. Dabei finden die in Abschnitt 2.9.2 beschriebenen Anpassungsoperationen bei Ausdrücken des Typs CHAR und ARRAY OF CHAR statt.
3. Werte eines Unterbereichs (siehe Abschnitt 2.12.2) können Variablen des Basistyps zugewiesen werden.
4. Ein ganzzahliger Wert kann immer einer reellen Variablen zugewiesen werden.
5. Alle anderen Zuweisungen sind fehlerhaft.

   Die erste Regel soll am Beispiel der Deklarationen aus Bild 32 verdeutlicht werden:

   ```
   MAT1:= MAT2        (zulässig)
   MAT4:= MAT1        (zulässig)
   MAT3:= MAT1        (unzulässig)
   MAT4:= MAT3        (unzulässig)
   ```

   Einige Compiler sind bei der Interpretation der Regeln zur Typkompatibilität etwas großzügiger. Sie erlauben in Einzelfällen auch Zuweisungen zwischen Variablen, die nur die gleiche Struktur besitzen MAT3:= MAT1 ist dann zulässig (siehe auch [3], Anhang E).

2.11 Prozeduren

Zerlegung in Teilprobleme

Eine besonders erfolgreiche Strategie beim Programmentwurf besteht darin, das Ausgangsproblem in geeignet gewählte kleinere Teilprobleme zu zerlegen. Ziel dieser Zerlegung ist es, überschaubare Programmteile zu erhalten, die mit dem restlichen Programm über wohldefinierte Schnittstellen kommunizieren, so daß die Korrektheit der Programmteile ohne Kenntnis der *Umgebung* gesichert werden kann. Als Beispiel ist in Bild 34 die Grobstruktur eines Programmes für ein Brettspiel mit dem Computer angegeben.

```
Ausgangsstellung aufbauen
REPEAT
  Spielstellung anzeigen
  Eingabe Spielerzug
  Spielstellung anzeigen
  Computerzug berechnen
UNTIL Spielende
Gewinner anzeigen
```

Bild 34: Teilprobleme eines Brettspiels

Schnittstellen zwischen Teilproblemen

Jede der im Klartext angegebenen Teilaufgaben läßt sich logisch von den anderen trennen. Ein Beispiel für eine Schnittstelle zwischen den Teilen ist die Spielstellung: Sie darf nur bei der Eingabe des Spielerzuges und des Computerzuges verändert werden. Die Teilaufgabe *Spielstellung anzeigen* kann hingegen auf die Spielstellung nur lesend zugreifen.
Pascal unterstützt diese Modularisierung des Programmes durch das Konzept der Prozeduren und Funktionen. Die Schnittstellen werden durch Parameterlisten und die Sichtbarkeitsregeln für Namen innerhalb des Programmtextes festgelegt.

Das Prozedurkonzept in Pascal

In Pascal würde man jede der obigen Teilaufgaben durch ein separates Programmstück, eine **Prozedur**, definieren. Jede Prozedur erhält in der Prozedurdeklaration einen Namen (Bezeichner). Im Anweisungsteil wird durch die Angabe des Namens der Prozedur die Ausführung der genannten Prozedur veranlaßt. Diese Art von Anweisungen nennt man Prozeduraufrufe.
Besonders nützlich sind Prozeduren, die von verschiedenen Stellen aufgerufen werden (z.B. *Spielstellung anzeigen* in Bild 34). Durch die Tatsache, daß die Prozedur nur einmal deklariert werden muß, spart man nicht nur Schreibarbeit und Speicherplatz, sondern man muß spätere Änderungen nur an einer Stelle durchführen.
Ein konkretes Beispiel soll die einfachste Form einer Prozedur vorstellen: Es ist der größte gemeinsame Teiler (ggT) der ganzen Zahlen A und B zu bestimmen. Diese Berechnung soll den ggT in der Variablen ERG ablegen. Die dazugehörige Prozedur GGT ist in Bild 35 angegeben:

```
PROCEDURE GGT;
(* Berechne ERG, den ggT der Zahlen A und B *)
BEGIN
  X:= A; Y:= B;
  WHILE X< >Y DO
    IF X>Y THEN X:= X-Y
           ELSE Y:= Y-X;
  ERG:= X;
END; (* GGT *)
```

Bild 35: Die Prozedur GGT

Deklaration einer Prozedur

Dies ist eine Prozedurdeklaration. Sie besteht aus dem Wortsymbol PROCEDURE und dem Prozedurnamen GGT, die zusammen den Prozedurkopf bilden. Zwischen den Wortsymbolen BEGIN und END steht der Anweisungsteil der Prozedur. Der Aufbau einer Prozedur ähnelt also der Struktur eines vollständigen Programmes.

Die Prozedurdeklarationen eines Programmes stehen am Ende des Vereinbarungsteils (siehe auch das allgemeine Programmschema in Bild 9). Dort werden alle Prozeduren, die das Hauptprogramm benutzt, in der Form von Bild 36 aufgeführt. In diesem Programm steht an zwei Stellen der Name GGT. Dort wird also die Prozedur GGT aufgerufen, um den ggT der Werte in den Variablen A und B zu berechnen. Der ggT ist bei der Rückkehr aus der Prozedur in der Variablen ERG gespeichert.

Prozeduraufruf

Beim Aufruf einer Prozedur werden alle Befehle im Anweisungsteil der Prozedur ausgeführt. Nach der Ausführung der letzten Anweisung der Prozedur (dort steht das Wortsymbol END) wird die Programmausführung mit dem nächsten Befehl im Hauptprogramm fortgesetzt. In den nächsten Abschnitten wird die Prozedur GGT noch mehrmals verändert und verbessert, um die weiteren Möglichkeiten von Pascal vorzustellen, die es erlauben, Prozeduren noch flexibler einzusetzen.

Beispiel: größter gemeinsamer Teiler

```
PROGRAM CALLGGT (OUTPUT);
  VAR A, B, ERG, X, Y: INTEGER;
  PROCEDURE GGT;
  BEGIN
    X:= A; Y:= B;
    WHILE X< >Y DO
      IF X>Y THEN X:= X-Y
             ELSE Y:= Y-X;
    ERG:= X;
```

Bild 36: Prozeduraufrufe

```
  END; (* GGT *)

BEGIN  (* Hauptprogramm *)
  A:= 9; B:= 3; GGT; WRITELN(ERG);
  A:= 8; B:= 3; GGT; WRITELN(ERG);
END.
```

Bild 36: Prozeduraufrufe (Schluß)

2.11.1 Lokalität von Namen

Lokale Variablen

In Bild 36 wurden die Variablen X und Y nur innerhalb der Prozedur GGT verwendet. Diese Zugehörigkeit kann man in Pascal dadurch unterstreichen, daß die Variablen innerhalb der Prozedur deklariert werden:

```
PROGRAM LOCALGGT (OUTPUT);
  VAR A, B, ERG: INTEGER;

  PROCEDURE GGT;
    VAR X,Y: INTEGER;
  BEGIN
    X:= A; Y:= B;
    WHILE X< >Y DO
      IF X>Y THEN X:= X-Y
             ELSE Y:= Y-X;
    ERG:= X;
  END; (* GGT *)

BEGIN  (* Hauptprogramm *)
  A:= 9; B:= 3; GGT; WRITELN(ERG);
  A:= 8; B:= 3; GGT; WRITELN(ERG);
  (* X:= 3  ist hier nicht möglich *)
END.
```

Bild 37: Lokale Variablen

Sichtbarkeitsregeln für Namen

Man bezeichnet X und Y als **lokale** Variablen der Prozedur GGT, da sie jetzt außerhalb der Prozedur nicht mehr sichtbar sind. Das heißt, die in Kommentarklammern angegebene Zuweisung X:=3 im Hauptprogramm würde der Compiler mit der Fehlermeldung

`104 Identifier not declared`

markieren. Durch diese lokale Deklaration nimmt eine Prozedur die Form eines eigenständigen Programmes an. Tatsächlich sind alle Deklarationen, die

im Hauptprogramm erlaubt sind, auch lokal möglich. Wenn Sie wieder einmal die Syntaxdiagramme im Anhang A zu Rate ziehen, werden Sie sehen, daß sich an den Programm- und den Prozedurkopf jeweils ein BLOCK anschließt. Das Syntaxdiagramm BLOCK enthält sowohl den Vereinbarungsteil als auch den Anweisungsteil.

Vorteile lokaler Deklarationen

Grundsätzlich versucht man, den Sichtbarkeitsbereich (engl. *scope*) eines Namens (Konstante, Variable etc.) möglichst klein zu halten. Diese Strategie wird manchmal auch als *Geheimnisprinzip* bezeichnet: Eine Prozedur *verbirgt* vor ihrer Umgebung nicht nur die Details ihres Anweisungsteils, sondern auch die lokalen Objekte.

Ein angenehmer Nebeneffekt dieses Prinzips der Lokalität ist eine Speicherplatzersparnis. Bei der Programmausführung wird erst beim Aufruf der Prozedur (GGT) Speicherplatz für die lokalen Objekte (die Variablen X und Y) reserviert. Am Ende der Ausführung der Prozedur wird dieser Speicherplatz wieder freigegeben und steht anderen Prozeduren zur Verfügung. Eine direkte Folge dieser Speicherorganisation ist, daß bei jedem neuen Aufruf einer Prozedur alle lokalen Variablen (wie die Variablen des Hauptprogrammes beim Programmstart) **undefiniert** sind.

Wird beim Aufruf einer Prozedur oder am Programmanfang festgestellt, daß nicht genügend Speicherplatz für die lokalen Variablen vorhanden ist, wird das Programm mit einer Fehlermeldung beendet:

```
STACK OVERFLOW ERROR
```

Globale Variablen und Seiteneffekte

Soll eine Variable ihren Wert zwischen zwei Aufrufen beibehalten, so muß man sie außerhalb der Prozedur *(global)* deklarieren. Deshalb wird die Variable G, die in Bild 38 die Anzahl der Aufrufe der Prozedur GLOBAL zählen soll, im Hauptprogramm deklariert.

```
PROGRAM GLOBALTEST(OUTPUT);
  VAR G: INTEGER;

  PROCEDURE GLOBAL;
  BEGIN
  G:= G+1; WRITELN('AUFRUF NUMMER', G);
  END; (* GLOBAL *)

BEGIN
  G:=0; GLOBAL; GLOBAL; GLOBAL;
END.
```

Bild 38: Globale Variable

In Bild 36 waren auch X und Y globale Variablen. Gerade in großen Programmen ist die Verwendung von globalen Variablen eine schwer zu entdeckende Fehlerquelle: Hätte man im Hauptprogramm in Bild 36 die Variablen X und Y benutzt, so würden als *Seiteneffekt* bei jedem Aufruf von GGT die Werte von X und Y zerstört werden.

Geschachtelte Prozeduren

Nachdem der Unterschied zwischen globalen und lokalen Objekten einer Prozedur bekannt ist, wird nun die **Schachtelung** von Prozeduren behandelt. Da alle Arten von Deklarationen in einer Prozedur erlaubt sind, kann eine Prozedur auch weitere Prozedurdeklarationen enthalten. Die Sichtbarkeitsregeln für lokale und globale Variablen gelten dann analog:

```
PROGRAM SCHACHTELUNG (OUTPUT);
  VAR A: REAL; B: REAL;

  PROCEDURE AUSSEN;
    VAR A: INTEGER;

    PROCEDURE INNEN;
      VAR I: INTEGER;
    BEGIN
      WRITELN('INNEN');
    END; (* INNEN *)

  BEGIN (* AUSSEN *)
    A:= 30; B:= 30;
    WRITELN('AUSSEN: A B =', A, B);
    INNEN; INNEN; INNEN;
  END; (* AUSSEN *)

BEGIN (* HAUPTPROGRAMM *)
  A:= 1; B:= 1;
  AUSSEN; AUSSEN;
  WRITELN('HAUPTPROGRAMM: A, B=', A,B);
END.
```

Bild 39: Geschachtelte Prozeduren

Lokale Prozeduren (wie die Prozedur INNEN) verwendet man aus dem gleichen Grund wie lokale Variablen: Die Prozedur INNEN wird nur innerhalb

der Prozedur AUSSEN benötigt. Deshalb sollte die Umgebung (in diesem Fall das Hauptprogramm) keinen Zugriff auf die lokale Prozedur besitzen. Konkret gesprochen ist es nicht möglich, die Prozedur INNEN im Anweisungsteil des Hauptprogrammes zu verwenden.

Formale Definition der Sichtbarkeitsregeln

Wie Ihnen sicher aufgefallen ist, werden in Bild 39 außerdem Variablen in verschiedenen Schachtelungsebenen deklariert, um die folgenden Regeln über die Sichtbarkeit von Namen in Pascal zu illustrieren:

1. Ein Name ist in dem Block P, in dem er deklariert wurde, sichtbar. Außerdem ist er in jedem Block sichtbar, der von P eingeschlossen wird, solange nicht Regel 2 gilt.
2. Eine Deklaration eines Namens X in einem Block macht alle Deklarationen des Namens X in äußeren Blöcken unsichtbar.
3. Die Standardnamen (wie z.B. READ, WRITE, SIN, INPUT, INTEGER) sind in einem imaginären Block deklariert, der das gesamte Programm umschließt.

Beispiele

Durch die Regel 1 ist die Variable B im Block SCHACHTELUNG, in der Prozedur AUSSEN und auch in der Prozedur INNEN sichtbar. Nach Regel 2 überdeckt die lokale Deklaration von A (INTEGER) die globale Variable A (REAL). In AUSSEN und INNEN ist also nur A (INTEGER) sichtbar. Aufgrund von Regel 3 sind z.B. die Standardbezeichner TRUE und FALSE im gesamten Programm sichtbar.

In der Prozedur AUSSEN finden sich folgende Zuweisungen:

```
A:= 30; B:= 30
```

Gemäß den obigen Sichtbarkeitsregeln wird damit der lokalen Variablen A und der globalen Variablen B der Wert 30 zugewiesen. Damit bleibt insbesondere der Wert der Variablen A (REAL) im Hauptprogramm unverändert.

Sollten Ihnen die Regeln (1) bis (3) etwas kompliziert erscheinen, so sollten Sie sich für den Augenblick nur merken, daß man in einer Prozedur Namen völlig unabhängig vom übrigen Programm wählen kann.

2.11.2 Parameter für Prozeduren

Universelle Prozeduren mit Parametern

Die Version der Prozedur GGT mit lokalen Variablen (Bild 37) ist immer noch nicht optimal in Pascal formuliert. Dort werden die Eingabewerte A und B sowie das Ergebnis ERG über globale Variablen übergeben. Durch die Benutzung von Parametern wird die Prozedur universeller und übersichtlicher:

```
PROGRAM PARAMETER (OUTPUT);
  VAR V: INTEGER;

  PROCEDURE GGT(A,B: INTEGER; VAR ERG: INTEGER);
```

Bild 40: GGT mit Parametern

```
  BEGIN
    WHILE A<>B DO
      IF A>B THEN A:= A-B
             ELSE B:= B-A;
    ERG:=A;
  END; (* GGT *)

BEGIN (* HAUPTPROGRAMM *)
  GGT(9, 3, V);
  WRITELN(V);
  GGT(17+4, 3, V);
  WRITELN(V);
END.
```

Bild 40: GGT mit Parametern (Schluß)

Formale und aktuelle Parameter

Der Prozedurkopf von GGT wird um eine Parameterliste erweitert. Dort werden die Übergabeparameter mit Namen und Typ angegeben. Die **formalen Parameter** A, B und ERG werden dadurch wie lokale Variablen der Prozedur GGT deklariert.

Beim Aufruf der Prozedur GGT müssen diesen formalen Parametern **aktuelle Parameter** übergeben werden. Dies geschieht in einer Parameterliste in Klammern hinter dem Prozeduraufruf. Beim Aufruf müssen Anzahl und Typ der aktuellen und formalen Parameter übereinstimmen. Es gibt zwei verschiedene Typen von Parametern, die beide in Bild 40 verwendet wurden.

Wertparameter entkoppeln aktuelle Parameter

Im Beispiel sind A und B **Wertparameter**. Sie verhalten sich innerhalb der Prozedur wie *normale* lokale Variablen. Jedoch werden sie zusätzlich beim Aufruf der Prozedur durch Ausdrücke (17+4 oder 3) als aktuelle Parameter initialisiert. Die Prozedur kann die Variablen A und B beliebig benutzen und ihnen auch Werte zuweisen, ohne daß in der aufrufenden Umgebung irgendeine Änderung bewirkt würde.

```
PROGRAM VALUEPARAMETER (OUTPUT);
  VAR AKTUELLERPARAMETER : INTEGER;

  PROCEDURE P (FORMALERPARAMETER: INTEGER);
  BEGIN
    WRITELN(FORMALERPARAMETER);
    FORMALERPARAMETER:= 0;
    WRITELN(FORMALERPARAMETER);
  END; (* P *)
```

Bild 41: Wertparameter (Fortsetzung nächste Seite) ►

```
BEGIN
  AKTUELLERPARAMETER:= 100;
  WRITELN(AKTUELLERPARAMETER);
  P (AKTUELLERPARAMETER);
  WRITELN(AKTUELLERPARAMETER);
END.
```

Bild 41: Wertparameter (Schluß)

Durch die Entkopplung von aktuellem und formalem Parameter konnten im Beispiel (Bild 40) die Hilfsvariablen X und Y eingespart werden. Wertparameter sind die vorherrschende Parameterart, da beliebige Ausdrücke als aktuelle Parameter auftreten können:

```
PROGRAM PARAMETERDEMO (OUTPUT);
  VAR I: INTEGER;

  PROCEDURE KASTEN (LAENGE, BREITE: INTEGER;
                    ZEICHEN       : CHAR);
  (* ZEICHNE EINEN KASTEN DIESER LAENGE UND BREITE, DER *)
  (* AUS DEM ZEICHEN GEBILDET WIRD.                     *)
    VAR I,J: INTEGER;
  BEGIN
    FOR I:= 1 TO BREITE DO
      BEGIN
        FOR J:= 1 TO LAENGE DO WRITE(ZEICHEN);
        WRITELN;
      END;
  END; (* KASTEN *)

BEGIN
  (* ZEICHNE EIN PAAR KÄSTEN: *)
  FOR I:= 1 TO 10 DO
    KASTEN (2*I,                  (* LAENGE  *)
            I+2,                  (* BREITE  *)
            CHR(I+ ORD('A')));    (* ZEICHEN *)
END.
```

Bild 42: Parameter verschiedener Typen

Variablenparameter zur Rückgabe von Werten

Wertparameter können keine Ergebnisse aus der Prozedur an die aufrufende Umgebung zurückliefern. In solchen Fällen benutzt man **Variablenparameter**. Hier ist der aktuelle Parameter immer eine Variable vom Typ des formalen Parameters. Variablenparameter werden bei der Deklaration in der Para-

meterliste von Wertparametern durch Voranstellen des Wortsymbols VAR unterschieden.
Die Prozedur GGT liefert das Ergebnis (den ggT) über den Variablenparameter ERG zurück. Deshalb muß beim Aufruf der dritte aktuelle Parameter immer eine Variable vom Typ INTEGER sein. Im Beispiel ist dies die Variable V.
Innerhalb der Prozedur ändert jede Zuweisung an einen formalen Variablenparameter unmittelbar den Wert des zugehörigen Parameters. Somit wird durch die Zuweisung ERG:=A (in Bild 40) der Variablen V der Wert A zugewiesen. Sie sollten sich den Unterschied nochmals dadurch verdeutlichen, daß Sie in Bild 41 den Prozedurkopf um das Wortsymbol VAR erweitern und das Programm kompilieren und ausführen:

```
PROCEDURE P (VAR FORMALERPARAMETER: INTEGER);
```

Der Aufruf mit Wertparametern wird auch als *call by value* bezeichnet. Den Aufruf mit Variablenparametern bezeichnet man dann als *call by reference*. Dies deutet bereits auf die Realisierung der Parameter auf dem Computer hin: Bei Wertparametern wird der Wert des Ausdruckes in den lokalen Speicherbereich der Prozedur kopiert (wie bei einer Zuweisung). Bei Variablenparametern wird hingegen nur ein Verweis (in Form einer Adresse) auf eine globale Variable übergeben.

Zusammengesetzte Typen als Parameter

Viele weitere Beispiele werden Sie in den nächsten Abschnitten finden. Am Ende dieses Abschnittes soll noch betont werden, daß alle Typen (auch zusammengesetzte) als Parameter auftreten können. Im Bild 43 werden zwei Arrays A und B mit je 5 Elementen elementweise addiert und das Ergebnis (5 Werte) in einem Array C gespeichert. Solche Arrays werden in der Mathematik als *Vektoren* bezeichnet, so daß die Summe von A und B eine *Vektorsumme* ist.

```
PROGRAM VEKTORSUMME (INPUT,OUTPUT);
  CONST N=5;
  TYPE VEKTOR = ARRAY [1..N] OF REAL;
  VAR X,Y,Z: VEKTOR;
      M    : ARRAY [1..N] OF VEKTOR; (* M ist eine Matrix! *)

  PROCEDURE HOLE(VAR A: VEKTOR);
  (* Vektor A einlesen *)
    VAR I: INTEGER;
  BEGIN
    FOR I:= 1 TO N DO READ (A[I]);
    READLN;
  END; (* HOLE *)
```

Bild 43: Vektorsumme (Fortsetzung nächste Seite) ►

```
  PROCEDURE DRUCKE(A: VEKTOR);
  (* A ausgeben *)
    VAR I: INTEGER;
  BEGIN
   FOR I:= 1 TO N DO WRITE (A[I],' ');
    WRITELN;
  END; (* DRUCKE *)

  PROCEDURE ADD(A,B: VEKTOR; VAR C: VEKTOR);
  (* C:= A + B komponentenweise *)
    VAR I: INTEGER;
  BEGIN
    FOR I:= 1 TO N DO
      C[I]:= A[I] + B[I];
  END; (* ADD *)

BEGIN
  HOLE(X); HOLE(Y);
  ADD(X,Y,Z);
  DRUCKE(X); DRUCKE(Y);
  WRITELN('Summe:');
  DRUCKE(Z);
  M[1]:= X; M[2]:= Z;
  ADD(M[1], M[2], M[3]);
  WRITELN('M[3] =');
  DRUCKE(M[3]);
END.
```

Bild 43: Vektorsumme (Schluß)

Typnamen in der Parameterliste zwingend

Hier werden also Vektoren von fünf Zahlen als Parameter übergeben. Aktuelle und formale Parameter müssen natürlich auch im Falle zusammengesetzter Variablen übereinstimmen. Dabei ist es wichtig, daß Sie einen Typ-**Namen** im Prozedurkopf angeben. Verboten ist also die Parameterliste

`PROCEDURE (A,B: ARRAY[1..N] OF REAL; VAR C: VEKTOR);`

2.11.3 Funktionen

Funktionen mit Ergebnissen

Neben den Variablenparametern gibt es in Pascal eine weitere Möglichkeit, Resultate aus Unterprogrammen zurückzuliefern. Diese Methode lehnt sich an die Notation von Funktionen in der Mathematik an. Dort schreibt man zum Beispiel

x = ggT (7,29)

Der Name der Funktion ggT repräsentiert also gleichzeitig den Wert der Berechnung. In Pascal definiert man solche Prozeduren, die nur einen skalaren Wert als Ergebnis liefern, als Funktionen:

```
PROGRAM FUNKTION(OUTPUT);
  VAR W: INTEGER;

  FUNCTION GGT(A,B: INTEGER): INTEGER;
  BEGIN
    WHILE A<>B DO
      IF A>B THEN A:= A-B
             ELSE B:= B-A;
    GGT:= A;
  END; (* GGT *)

BEGIN (* HAUPTPROGRAMM *)
  W:= GGT(12345, 25325);
  WRITELN(W, GGT(9,3));
  W:= W + GGT( GGT(1234, 432), 444);
END.
```

Bild 44: Der ggT als Funktion

Deklaration

Wenn Sie Bild 44 mit Bild 40 vergleichen, so werden Sie folgende Unterschiede zwischen einer Funktion und einer »normalen« Prozedur feststellen: Im Prozedurkopf wird das Wortsymbol PROCEDURE durch FUNCTION ersetzt, und am Ende der Parameterliste steht zusätzlich nach einem Doppelpunkt der **Ergebnistyp** der Funktion. An dieser Stelle muß (wie bei einer Parameterliste) ein Typ-**Name** stehen:

Richtig!

```
TYPE SHORTSTRING = STRING[16];
FUNCTION REVERSE(S: SHORTSTRING): SHORTSTRING;
```

also nicht:

Falsch!

```
FUNCTION REVERSE(S: STRING[16]): STRING[16];
```

Erweiterung in Pascal 2.0

Als Ergebnistyp einer Funktion sind im *report* nur skalare Typen und Zeiger (siehe Abschnitt 2.18) erlaubt, während *Pascal 2.0* auch beliebige zusammengesetzte Typen als Funktionsergebnis unterstützt.

Die exakte Syntax einer Prozedur- und Funktionsdeklaration ist natürlich wieder dem Anhang A (Syntaxdiagramm BLOCK und PARAMETER LISTE) zu entnehmen.

Innerhalb der Funktion muß das Funktionsergebnis an mindestens einer Stelle durch eine Zuweisung an den Funktionsnamen festgelegt werden. In der Funktion GGT geschieht dies mit der Zuweisung GGT:=A. GGT ist also nicht nur

der Name der Funktion, sondern auch der Name für das Ergebnis der Funktion. Die Anweisungen im Hauptprogramm in Bild 44 zeigen, daß man Funktionsaufrufe wie in der Mathematik üblich beliebig geschachtelt innerhalb von Ausdrücken verwenden kann.

Funktionsaufruf

Funktionsaufrufe sind nicht wie Prozeduraufrufe als eigenständige Anweisungen möglich:

Richtig! `W:= GGT(3,4)`

Falsch! `BEGIN GGT(3,4) END`

Nützliche Funktionen

Das folgende Programm beinhaltet mehrere Funktionsdeklarationen, die recht häufig in alltäglichen Programmen verwendet werden.

```
PROGRAM FUNCTIONS (INPUT,OUTPUT);
  CONST VEKTORLAENGE = 3;
  TYPE  VEKTOR       = ARRAY [1..VEKTORLAENGE] OF INTEGER;
  VAR   V            : VEKTOR;
        S            : STRING;
        I            : INTEGER;
  FUNCTION MAX(A,B: INTEGER): INTEGER;
  (* BESTIMME DAS MAXIMUM DER BEIDEN ZAHLEN *)
  BEGIN
    IF A>B THEN MAX:=A
           ELSE MAX:=B;
  END; (* MAX *)

  FUNCTION INTRANDOM(LIMIT: INTEGER): INTEGER;
  (* GANZE ZUFALLSZAHL ZWISCHEN 1 UND LIMIT ERZEUGEN *)
  BEGIN
    INTRANDOM:= 1 + INT(RANDOM(0) * LIMIT);
  END; (* INTRANDOM *)

  FUNCTION FAKULTAET (N: INTEGER): REAL;
  (* N ! = 1 * 2 * 3 * ... * N BERECHNEN *)
    VAR P: REAL;
  BEGIN
    P:=1;
     WHILE N>1 DO
      BEGIN P:= P*N; N:= N-1; END;
    FAKULTAET:= P;
  END; (* FAKULTAET *)
```

Bild 45: Beispiele für Funktionen

```
  FUNCTION CENTER (S: STRING; FIELD: INTEGER): STRING;
  (* ZENTRIERE S IN EINEM STRING DER LAENGE FIELD *)
  BEGIN
    WHILE LENGTH(S)<FIELD-1 DO
      S:= ' ' + S + ' ';
    IF LENGTH(S)=FIELD-1 THEN CENTER:= S + ' '
                         ELSE CENTER:= S;
  END; (* CENTER *)

  FUNCTION BETRAG(V: VEKTOR): REAL;
  (* BERECHNE DEN BETRAG (EUKLIDISCHE NORM / LAENGE) DES *)
  (* VEKTORS V                                           *)
    VAR I     : INTEGER;
        SIGMA: REAL;
  BEGIN
    SIGMA:= 0.0;
    FOR I:= 1 TO VEKTORLAENGE DO
      SIGMA:= SIGMA + SQR(V[I]);
    BETRAG:= SQRT(SIGMA);
  END; (* BETRAG *)

BEGIN (* HAUPTPROGRAMM *)
   FOR I:= 1 TO VEKTORLAENGE DO
    BEGIN
      V[I]:= INTRANDOM(200); WRITE(V[I]:4);
    END;
  WRITELN
  WRITELN('Das Maximum ist:', MAX(MAX(V[1],V[2]),
  V[3]));
  WRITELN('Der Betrag ist:',  BETRAG(V));
  S:= CENTER('*', 40);
  WRITELN( CENTER('Dieser Text wird auf ',40));
  WRITELN( CENTER('40 Spalten zentriert', 40));
  WRITELN; WRITELN(S);
  FOR I:= 0 TO 15 DO
    WRITELN(I:5, '! =', FAKULTAET(I));
END.
```

Bild 45: Beispiele für Funktionen (Schluß)

2.11.4 Standardprozeduren

Bereits bei Ihren ersten Schritten in Pascal hatten Sie die Anweisungen WRITE und READ sowie arithmetische Funktionen wie SIN, COS und ABS verwendet.

Vordefinierte Prozedur- und Funktionsnamen

Im Unterschied zu den Wortsymbolen (BEGIN, END, WHILE, OR, MOD ...) handelt es sich hierbei um vordefinierte Namen für Prozeduren und Funktionen. Aus den in Abschnitt 2.11.1 beschriebenen Sichtbarkeitsregeln für Namen geht hervor, daß diese Standardbezeichner auch durch benutzerdefinierte Namen *verdeckt* werden können. Diese Tatsache ist besonders wichtig, wenn es um die Verwendung von Namen geht, die nur in *Pascal 2.0* eine vordefinierte Bedeutung besitzen: Falls Sie z.B. aus der Literatur ein Programm übernehmen wollen, das den Namen CIRCLE verwendet, so können Sie nicht in Konflikte mit der Prozedur CIRCLE von *Pascal 2.0* geraten, da die Prozedur CIRCLE in einem imaginären Block deklariert ist, der das gesamte Programm umgibt.

Änderung der Standardnamen durch den Benutzer

Als Beispiel zeigt Bild 46 eine Deklaration der Quadratwurzelfunktion SQRT. Durch die Vereinbarung der Funktion SQRT wird die Standardfunktion SQRT *verdeckt*, so daß spätere Aufrufe der Funktion SQRT nur die benutzerdefinierte Funktion aktivieren:

```
PROGRAM NEWSQRT (INPUT,OUTPUT);
  CONST EPS = 1.0E-7;   (* Genauigkeit *)
  VAR   X: REAL;

  FUNCTION SQRT(X: REAL): REAL;
    VAR Y,Z: REAL;
  BEGIN
    IF X<0 THEN
      BEGIN
        WRITELN('Fehler in SQRT'); HALT;
      END
    ELSE
      BEGIN
        Y:= 2; (* Startwert Z berechnen *)
        REPEAT
          Z:= Y; Y:= Y*Y
        UNTIL Y>X;
        REPEAT   (* Iteration *)
          Y:= Z; Z:= 0.5 * (Y + X/Y)
        UNTIL ABS(Y-Z)<= EPS;
     END;
```

Bild 46: Deklaration der Quadratwurzelfunktion

```
    SQRT:= Z;
  END; (* SQRT *)

BEGIN
  X:= 3.0;
  REPEAT
    WRITELN(X:13, SQRT(X):13); X:= 10 * X;
  UNTIL X>1E9;
END.
```

Bild 46: Deklaration der Quadratwurzelfunktion (Schluß)

Quadratwurzelberechnung variabler Genauigkeit

Durch eine Erniedrigung der Genauigkeit EPS läßt sich bei Bedarf die Geschwindigkeit der Routine erhöhen. Die Eigenschaft mancher vordefinierter Prozeduren, eine **variable** Anzahl von Parametern zu besitzen (z.B. WRITE, READ), läßt sich mit benutzerdefinierten Prozeduren nicht simulieren.
Die Wirkung jeder einzelnen Standardprozedur und -funktion wird in Kapitel 4.4.4 erläutert.

2.11.5 Rekursion

In Abschnitt 2.11.1 über die Schachtelung von Prozeduren wurde erklärt, daß in einem Block jede Prozedur aufgerufen werden kann, deren Name sichtbar ist. Dies bedeutet, daß sich eine Prozedur auch **selbst** aufrufen kann. Dieser Vorgang wird **Rekursion** genannt.

Speicherverwaltung bei rekursiven Prozeduraufrufen

Durch die Tatsache, daß bei jedem Aufruf einer Prozedur Speicherplatz für die lokalen Objekte bereitgestellt wird, werden bei rekursiven Aufrufen einer Prozedur P verschiedene *Inkarnationen* der lokalen Variablen von P erzeugt. Wegen dieser Methode der Speicherverwaltung werden beim rekursiven Aufruf einer Prozedur die *alten* Werte der lokalen Variablen nicht überschrieben. Dies läßt sich am besten mit einem einfachen Beispielprogramm verdeutlichen (Bild 47).

```
PROGRAM REKURSION (OUTPUT);
  PROCEDURE REKURSIV (N: INTEGER);
    VAR LOKAL: INTEGER;
  BEGIN
    LOKAL:= N;
    WRITELN (' ': N, 'LOKAL =', LOKAL);
    IF N<4 THEN REKURSIV(N+1);          (* <<-------- *)
    WRITELN (' ': N, 'LOKAL =', LOKAL);
```

Bild 47: Eine einfache Rekursion (Fortsetzung nächste Seite)

```
  END; (* REKURSIV *)
BEGIN
  REKURSIV(1);
END.
```

Bild 47: Eine einfache Rekursion (Schluß)

Rekursive Einrückung eines Textes

Die Prozedur REKURSIV speichert zunächst (zu Demonstrationszwecken) den Wert des Parameters N in einer lokalen Variablen (LOKAL). Dieser Wert wird nun in zwei identischen Ausgabeanweisungen um N Stellen eingerückt gedruckt. Die eigentlich interessante Anweisung ist mit einem Pfeil markiert: Ist beim Aufruf der Prozedur der Parameter noch nicht gleich 4, so ruft sich die Prozedur selbst auf. Als Parameter für diesen Selbstaufruf wird die Zahl N+1 verwendet.

Da jeder Aufruf der Prozedur REKURSIV seine *eigenen* Variablen N und LOKAL besitzt, wird durch diesen rekursiven Aufruf der Inhalt der Variablen N und LOKAL in der aufrufenden Umgebung **nicht** verändert. Deshalb ergibt sich folgende Ausgabe:

```
LOKAL = 1
 LOKAL = 2
  LOKAL = 3
   LOKAL = 4
   LOKAL = 4
  LOKAL = 3
 LOKAL = 2
LOKAL = 1
```

Jeweils zwei Zeilen, die gleich weit eingerückt sind, stammen von derselben *Inkarnation* der Prozedur REKURSIV. Dieses Programm ist zwar nicht sehr sinnvoll, zeigt aber deutlich die *geschachtelten* Aufrufe:

```
REKURSIV(1)   ruft
  REKURSIV(2)   ruft
    REKURSIV(3)   ruft
      REKURSIV(4)
```

Beenden einer Rekursion

Außerdem sehen Sie, daß diese Folge rekursiver Aufrufe irgendwann beendet werden muß. Deshalb werden rekursive Aufrufe immer durch eine Bedingung kontrolliert. Im Beispiel aus Bild 47 ist dies die IF-Anweisung IF N < 4 THEN.

Schon jetzt möchte ich Sie davor warnen, rekursive Prozeduren Schritt für Schritt nachzuvollziehen, indem Sie sich die Werte aller lokalen Variablen notieren und so den Programmablauf zu analysieren versuchen. Dabei werden Sie nach wenigen Schritten schon an einem heillosen Durcheinander verzweifeln. Vielmehr muß man rekursive Prozeduren *deklarativ* verstehen. So kann man z.B. die Prozedur REKURSIV wie folgt beschreiben:

Deklarative Analyse des Programms in Bild 47

1. Drucke N Stellen eingerückt die Zahl N
2. Ist der Text noch nicht vier Stellen eingerückt, so drucke einen Block, der N+1 Stellen eingerückt ist.
3. Drucke N Stellen eingerückt die Zahl N

Um Ihr *deklaratives* Verständnis zu trainieren, sollten Sie jetzt ein Blatt Papier zur Hand nehmen und die Ausgabe notieren, die Sie beim Aufruf REKURSIV2(1) in Bild 48 erwarten. Im Unterschied zur Prozedur REKURSIV druckt REKURSIV2 den eingerückten Block zweimal. (Ein Tip: Die Ausgabe ist genau 30 Zeilen lang!)

```
PROGRAM REKURSION2 (OUTPUT);
  PROCEDURE REKURSIV2 (N: INTEGER);
    VAR LOKAL: INTEGER;
  BEGIN
    LOKAL:= N;
    WRITELN (' ': N, 'LOKAL =', LOKAL);
    IF N<4 THEN REKURSIV2(N+1);          (* <<-------- *)
    IF N<4 THEN REKURSIV2(N+1);          (* <<-------- *)
    WRITELN (' ': N, 'LOKAL =', LOKAL);
  END; (* REKURSIV *)
BEGIN
  REKURSIV2(1);
END.
```

Bild 48: Rekursion (2)

Das Prinzip der Rekursion

Das Prinzip der Rekursion besteht darin, daß man zur Lösung einer großen Aufgabe die Lösung der Aufgabe für *kleinere* Werte benutzt. Diese etwas tautologisch anmutende Strategie soll ein kleines grafisches Beispielprogramm illustrieren:

```
PROGRAM DRAWCIRCLES (INPUT,OUTPUT, GRAPHIC);
  CONST XMAX = 319;
        YMAX = 199;
  VAR   T    : INTEGER;

  PROCEDURE QUADCIRCLE (TIEFE: INTEGER;
                        MX, MY, R : INTEGER);
  (* ZEICHNE REKURSIV EINEN KREIS, DER VIER KREISE *)
  (* ENTHÄLT, DIE SICH GEGENSEITIG BERÜHREN.       *)
    VAR R2: INTEGER;
  BEGIN
```

Bild 49: Rekursives Grafikprogramm (Fortsetzung nächste Seite) ►

```
      CIRCLE (1, MX, MY, R,,0,360,0,10);
      IF TIEFE>1 THEN
        BEGIN (* Bilde einen Kreis aus 4 kleineren Kreisen *)
          R2:= R DIV 2;
          QUADCIRCLE(TIEFE-1, MX-R2, MY    ,R2);
          QUADCIRCLE(TIEFE-1, MX   , MY-R2 ,R2);
          QUADCIRCLE(TIEFE-1, MX+R2, MY    ,R2);
          QUADCIRCLE(TIEFE-1, MX   , MY+R2, R2);
        END;
      (* --- Punkt 1 -- *)
    END; (* QUADCIRCLE *)

  BEGIN
    COLOR(0,1); COLOR(1,2); COLOR(5,5);
    WRITE('Welche Tiefe? :4' #157);
    READLN(T);
    GRAPHIC(1,1);
    QUADCIRCLE(T, XMAX DIV 2, YMAX DIV 2, YMAX DIV 2);
    REPEAT UNTIL KEYPRESSED;
    GRAPHIC(0);
  END.
```

Bild 49: Rekursives Grafikprogramm (Schluß)

Rekursive Muster am Bildschirm

Dieses Programm zeichnet ein Muster aus geschachtelten Kreisen. Ein Muster der Tiefe N wird durch folgende (rekursive) Regel beschrieben:

1. Zeichne einen Kreis
2. Ist die Tiefe N größer als 1, so zeichne das Muster viermal (links, oben, rechts und unten) in halber Größe mit der Tiefe N–1.

Exponentielle Laufzeiten

Wenn Sie das Programm nacheinander für die Tiefen 1, 2, 3 und 4 ausführen lassen, werden Sie das obige Konstruktionsprinzip sehr deutlich erkennen. Außerdem werden Sie feststellen, daß durch Rekursion die Programmlaufzeit nach dem Schneeballsystem exponentiell ansteigen kann:

Tiefe	Anzahl der Kreise	
1	1	= 1
2	1 + 4	= 5
3	1 + 4 (1 + 4)	= 21
4	1 + 4 (1 + 4 (1 + 4))	= 85
5	1 + 4 (1 + 4 (1 + 4 (1 + 4)))	= 341

Weitere Muster

Grafische Programme sind sehr gut geeignet, das Prinzip der Rekursion anschaulich darzustellen, da optisch die *Selbstbezüglichkeit* einer Struktur unmittelbar zu erkennen ist. Eine Rekursion wie in Bild 49 läßt sich auch »von hinten« aufzäumen, indem man zunächst die Muster niedriger Tiefe zeichnet. Sie sollten deshalb zum Vergleich die Anweisung

```
CIRCLE (1, MX, MY, R,,0,360,0,10);
```

an die Stelle (* --- Punkt1 --- *) stellen. Zwar ist das endgültige Muster unverändert, jedoch werden Sie erkennen, daß die Zeichenstrategie sich geändert hat. Auf der mitgelieferten Programmdiskette findet sich das Programm »TRIA.P«. Es arbeitet nach dem gleichen Rekursionsschema wie das Programm in Bild 49, um geschachtelte Dreiecke wählbarer Verzerrung zu zeichnen.

Permutation eines Wortes

Nachdem Sie einmal das Prinzip der Rekursion verstanden haben, werden Sie eine Vielzahl von Problemlösungen durch rekursive Algorithmem formulieren. Ein sinnvolles Beispiel für die Verwendung der Rekursion zeigt Bild 50. Das Programm soll alle möglichen Anordungen (Permutationen) eines Strings mit N Zeichen Länge drucken. Für den String 'ABC' mit drei Zeichen gibt es diese 6 Anordnungen:

ABC BAC CBA BCA ACB CAB

Deklarative Problemlösung

Der rekursive Algorithmus zur Ausgabe lautet folgendermaßen:

1. Ist die Länge N gleich 1, so gibt es nur diese Anordnung. Drucke diese Anordnung.
2. Ansonsten drucke alle Möglichkeiten, die ersten (N–1) Zeichen im String anzuordnen, ohne das N-te Zeichen zu verändern.
   Für alle Positionen i von 1 bis N–1: Tausche das i. Zeichen mit dem letzten (N-ten) Zeichen im String. Drucke alle Möglichkeiten für diese Anordnung. Mache die Vertauschung rückgängig.

So formuliert läßt sich der Algorithmus direkt in ein Pascal-Programm umsetzen (siehe Bild 50).

```
PROGRAM ANORDNUNGEN (INPUT,OUTPUT);
  VAR I: INTEGER;
      A: STRING;

  PROCEDURE ANORDNUNG(S: STRING; N: INTEGER);
  (* Drucke alle Anordnungen der ersten N Zeichen im  *)
  (* String S, ohne die übrigen Zeichen zu verändern. *)
     VAR C: CHAR;
         I: INTEGER;
  BEGIN
    IF N=1 THEN WRITE(S,' ')
```

Bild 50: Permutationen (Fortsetzung nächste Seite) ►

```
    ELSE
      BEGIN
        ANORDNUNG (S,N-1);
        FOR I:= 1 TO N-1 DO
          BEGIN
            C:= S[I]; S[I]:= S[N]; S[N]:= C;
            ANORDNUNG(S,N-1);
            C:= S[I]; S[I]:= S[N]; S[N]:= C;
          END;
      END;
  END; (* ANORDNUNG *)

BEGIN
  READLN(A);
  ANORDNUNG (A, LENGTH(A));
  WRITELN;
END.
```

Bild 50: Permutationen (Schluß)

»Beweis« der Korrektheit rekursiver Algorithmen

Zugegebenermaßen erfordern rekursive Algorithmen eine gewisse geistige Leistung, da es bei ihnen im allgemeinen nicht mehr möglich ist, Anweisung für Anweisung den Programmablauf zu verfolgen. Vielmehr muß man eine rekursive Prozedur folgendermaßen »verstehen«:

1. Für Strings der Länge 1 arbeitet die Prozedur ANORDNUNG offensichtlich korrekt.
2. Angenommen, die Prozedur ANORDNUNG arbeitet für Strings der Länge N–1 korrekt, so werden durch die FOR-Anweisung auch alle Anordnungen der Länge N korrekt erzeugt.

Aus 1 und 2 zusammen folgt, daß für Strings beliebiger Länge alle Anordnungen bestimmt werden.

Iteration versus Rekursion

Natürlich gibt es auch Fälle, in denen eine Rekursion unnötig ist, da eine gleichwertige *iterative* Lösung existiert. Man könnte z.B. die Funktionen CENTER und FAKULTAET aus Bild 45 auch folgendermaßen rekursiv definieren:

```
FUNCTION FAKULTAET(N: INTEGER): REAL;
(* N ! = 1 * 2 * 3 * ... * N BERECHNEN *)
BEGIN
  IF N=0 THEN FAKULTAET:= 1.0
         ELSE FAKULTAET:= N * FAKULTAET(N-1);
END; (* FAKULTAET *)
```

Bild 51: Unnötige Rekursionen

```
FUNCTION CENTER (S: STRING; FIELD: INTEGER): STRING;
(* ZENTRIERE S IN EINEM STRING DER LAENGE FIELD *)
BEGIN
  IF LENGTH(S)<FIELD-1 THEN
    CENTER:= CENTER(' ' + S + ' ')
  ELSE
    IF LENGTH(S)=FIELD-1 THEN
       CENTER:= S + ' '
    ELSE
       CENTER:= S;
END; (* CENTER *)
```

Bild 51: Unnötige Rekursionen (Schluß)

Jedoch erfordert jeder rekursive Aufruf, abgesehen von dem Speicherplatz für die lokalen Variablen, auch einen gewissen Verwaltungsaufwand, der bei zusammengesetzten Anweisungen (FOR-, WHILE- und REPEAT-Schleifen) nicht auftritt.

Schnelles Sortieren mit QUICKSORT

Zum Abschluß der Ausführungen über Rekursion wird der in Kapitel 2.9.1 angekündigte verbesserte Sortieralgorithmus vorgestellt. Die Aufgabe ist wiederum, ein Array A mit N (ganzen) Zahlen aufsteigend zu sortieren.

Die Idee zu einer rekursiven Lösung besteht darin, das Array A in zwei Teile A[1..K] und A[K+1..N] aufzuteilen, so daß jeder Teil für sich, ohne Kenntnis der Zahlen im anderen Teil, sortiert werden kann. Diese Zerlegung und den Index K findet man mit der folgenden Strategie:

Zerlegung des Arrays in zwei Hälften

1. Man wählt einen (zufälligen) Wert X aus dem Array. Dies kann zum Beispiel der Wert in der Mitte des Arrays sein.
2. Anschließend bestimmt man den Index K, so daß die *linke* Hälfte A[1..K] des Arrays nur Werte kleiner oder gleich X enthält, während die *rechte* Hälfte A[K+1..N] nur Werte größer oder gleich X enthält.

Sortieren der beiden Hälften

3. Sortiert man anschließend beide Teilarrays, die mindestens ein Element weniger als das ursprüngliche Array enthalten, so ist schließlich das gesamte Array A[1..N] sortiert.

Ein Zahlenbeispiel

Die Tatsache, daß in jedem Schritt die Arraygröße um mindestens ein Element sinkt, ist wichtig, damit die Rekursion auch korrekt terminiert. Betrachten Sie diese Strategie an einem Zahlenbeispiel:

8 5 7 6 3 4 8 4

Da acht Elemente vorliegen, wählt man in Schritt 1 das 4. Element (X=6) im Array. Die Zerlegung in zwei Teile sieht nach dem 2. Schritt wie folgt aus:

4 5 4 3 6 7 8 8

Links und rechts von der Zahl 6 sind (in beliebiger Reihenfolge) die Zahlen kleiner bzw. größer als 6 aufgeführt. Da es vier Zahlen kleiner als 6 gibt, besitzt

der Index K den Wert K= 4 + 1 = 5. Im dritten Schritt werden nun die Teilarrays getrennt sortiert

3 4 4 5 6 7 8 8

Effizienter Algorithmus zur Zerlegung

Damit ist die gesamte Folge sortiert. Für Schritt 2 (die Zerlegung in zwei Teilarrays) muß noch ein effizienter Algorithmus angegeben werden:

2.1 Setze zwei Zeiger I und J auf das erste und letzte Element im Array

2.2 Lasse I nach *rechts* wandern, bis es auf ein Element zeigt, das größer als X ist.
Lasse J nach *links* wandern, bis es auf ein Element zeigt, das kleiner als X ist.

2.3 Da A[I] und A[J] jeweils auf der *falschen* Seite von X stehen, tausche beide Elemente aus.

2.4 Wiederhole diese Schritte, bis sich beide Zeiger I und J treffen.

Der skizzierte Algorithmus wurde von seinem *Erfinder* C.A.R.Hoare **Quicksort** getauft und findet sich in Bild 52 als ein vollständiges Pascal-Programm. Um die Korrektheit für alle Belegungen des Arrays A zu sichern, ist die exakte Formulierung der Bedingungen in den Schleifen nötig. Die Diskussion solcher Details und die Berechnung der Rechenzeit finden Sie in [2] (siehe Anhang E). Die Sortierung erfolgt in der rekursiven Prozedur QUICK, deren Parameter L und R den Index des ersten und letzten Elementes im zu sortierenden Array A enthalten. In Bild 52 sind zur Verdeutlichung die Nummern der obigen Schritte als Kommentare angegeben.

```
PROGRAM SORTIEREN (INPUT,OUTPUT);
  CONST N= 16;
  TYPE  ELEMENT = INTEGER;
  VAR   A: ARRAY [1..N] OF ELEMENT;

  PROCEDURE ERZEUGEN;
    VAR I: INTEGER;
  BEGIN
    WRITELN('Geben Sie eine unsortierte Folge');
    WRITELN('von ', N, ' ganzen Zahlen an:');
    FOR I:= 1 TO N DO READ(A[I]);
    READLN;
  END; (* ERZEUGEN *)
  PROCEDURE AUSGEBEN;
    VAR I: INTEGER;
  BEGIN
    WRITELN('Die Folge lautet:');
```

Bild 52: Quicksort

```
    FOR I:= 1 TO N DO WRITE(A[I]:5);
    WRITELN;
  END; (* AUSGEBEN *)

  PROCEDURE QUICK (L,R: INTEGER);
  (* Sortiere das Array im Bereich A[L..R] *)
    VAR I, J: INTEGER;
        X, Y: ELEMENT;
  BEGIN
    X:= A[(L+R) DIV 2];                        (* 1.   *)
    I:=L; J:=R;
    REPEAT
      WHILE A[I]<X DO I:= I+1;                 (* 2.1  *)
      WHILE A[J]>X DO J:= J-1:            (* 2.2  *)
      IF I<=J THEN
        BEGIN
          Y:= A[I]; A[I]:= A[J]; A[J]:= Y;     (* 2.3  *)
          I:= I+1; J:= J-1;
        END;
    UNTIL I>J;                                 (* 2.4  *)
    IF I<R THEN QUICK(I,R);                    (* 3    *)
    IF L<J THEN QUICK(L,J);                    (* 3    *)
  END; (* QUICK *)

BEGIN
  ERZEUGEN; AUSGEBEN;
  QUICK(1,N);            (* vom 1. bis zum N. Element *)
  AUSGEBEN;
END.
```

Bild 52: Quicksort (Schluß)

2.11.6 FORWARD-Deklarationen

In sehr speziellen Anwendungen kann eine indirekte Rekursion der Prozeduren A, B, C in der folgenden Form notwendig sein:

Indirekte Rekursion

A ruft B
A ruft C
C ruft B
B ruft C

In diesem Fall läßt sich keine Reihenfolge der Prozedurdeklarationen finden, in der die Deklaration jeder Prozedur vor ihrer Anwendung steht. Für diese

zyklischen Referenzen existiert in Pascal die FORWARD-Deklaration von Prozeduren und Funktionen:

```
PROGRAM FORWARDPROCEDURES (OUTPUT);
  PROCEDURE B(I: INTEGER); FORWARD;

  PROCEDURE C(X: CHAR);
  BEGIN
    IF X = '+' THEN B(ORD(X));
  END; (* C *)

  PROCEDURE B;
  BEGIN
    IF CHR(I)<>'+' THEN C(CHR(I));
  END; (* B *)

  PROCEDURE A;
  BEGIN
    B(13); C('+');
  END; (* A *)

BEGIN
  A;
END.
```

Bild 53: Beispiel für FORWARD

Deklaration des Programmkopfs getrennt vom Rumpf

Indem man nur den Kopf der Prozedur B, gefolgt von dem Wortsymbol FORWARD nennt, kann man B in der Prozedur C verwenden, obwohl der eigentliche Anweisungsteil der Prozedur B erst hinter der Prozedur C steht. Der Anweisungsteil jeder FORWARD-deklarierten Prozedur muß nachträglich hinter einem Prozedurkopf **ohne Parameterliste** aufgeführt werden. Ansonsten meldet der Compiler

```
119   UNSATISFIED 'FORWARD' DECLARATION IN PROGRAM
```

Aufgaben

Standardlösungen als Prozeduren

1. Formulieren Sie einige der Lösungen der Aufgaben früherer Abschnitte als Prozeduren oder Funktionen. Überlegen Sie, welche Parameter diese Prozeduren benötigen. Beachten Sie dabei die Unterschiede zwischen Variablen- und Wertparametern. Diese Prozeduren können Sie bei Bedarf als INCLUDE-Files (siehe Abschnitt 4.4.6.4) auf einer Diskette speichern und später in eigenen Programmen verwenden. Beispiele:

```
PROCEDURE BLOCKSATZ(VAR S: STRING; N: INTEGER;
                    RECHTS: BOOLEAN);
(* Formatiere die Textzeile in S im Blocksatz auf *)
(* N Spalten. Falls RECHTS=TRUE, werden Leer-     *)
(* stellen von rechts, ansonsten von links be-    *)
(* ginnend in den Wortzwischenräumen eingefügt.   *)
CONST N      = ...
TYPE ELEMENT = ...
     FELD = ARRAY [1..N] OF ELEMENT;
PROCEDURE BUBBLESORT(VAR A: FELD);
(* Sortiere das Feld A aufsteigend mit dem Algo- *)
(* rithmus BUBBLESORT.                           *)
FUNCTION POSITION(VAR A: FELD; W: ELEMENT): INTEGER;
(* Suche das Element W im aufsteigend sortierten *)
(* Array A. Das Funktionsergebnis ist der Index  *)
(* im Intervall 1..N. Wird W nicht gefunden, so  *)
(* ist das Funktionsergebnis die Zahl 0.         *)
```

Solche kommentierten Prozedurrümpfe sind eine geeignete Methode, um beim Entwurf größerer Programme die Teilaufgaben überschaubar aufzuteilen. Andererseits sind sie ein gutes Dokumentationshilfsmittel, um die Struktur eines bestehenden Programmes zu erläutern, da man die Details des Anweisungsteils einer Prozedur oder Funktion bei Bedarf ignorieren kann.

Balken- und Tortendiagramm

2. Zur Darstellung von Meßwerten und statistischen Daten sind *Balkendiagramme* und *Tortendiagramme* besonders geeignet. Entwickeln Sie Prozeduren, die möglichst allgemein verwendbare beschriftete Balken- und Tortendiagramme zeichnen. Bei Bedarf können Sie die Diagramme auch dreidimensional ausgeben. Siehe hierzu Bild 54, Seite 132.
   Vielleicht helfen Ihnen beim Programmieren die folgenden Prozedurrümpfe und Anmerkungen:

Beschriftung einer Grafik

   Besonders problematisch ist die Beschriftung von Grafiken, da der C 128 auch im Grafikmodus bei der Ausgabe von Texten an ein 40∗25-Zeichen-Raster gebunden ist. Damit ist jedes Zeichen auf dem Grafikbildschirm 8∗8 Pixel groß. Dementsprechend ist es sinnvoll, Grafiken an dieses 8∗8-Raster anzupassen.

```
CONST MAXTEILE = 12;
      ZEICHENHOEHE  = 8; (* Pixel pro Zeichen *)
      ZEICHENBREITE = 8; (* Pixel pro Zeichen *)
TYPE  TWERTE    = ARRAY[1..MAXTEILE] OF REAL;
      TLEGENDE  = ARRAY[1..MAXTEILE] OF STRING[8];
```

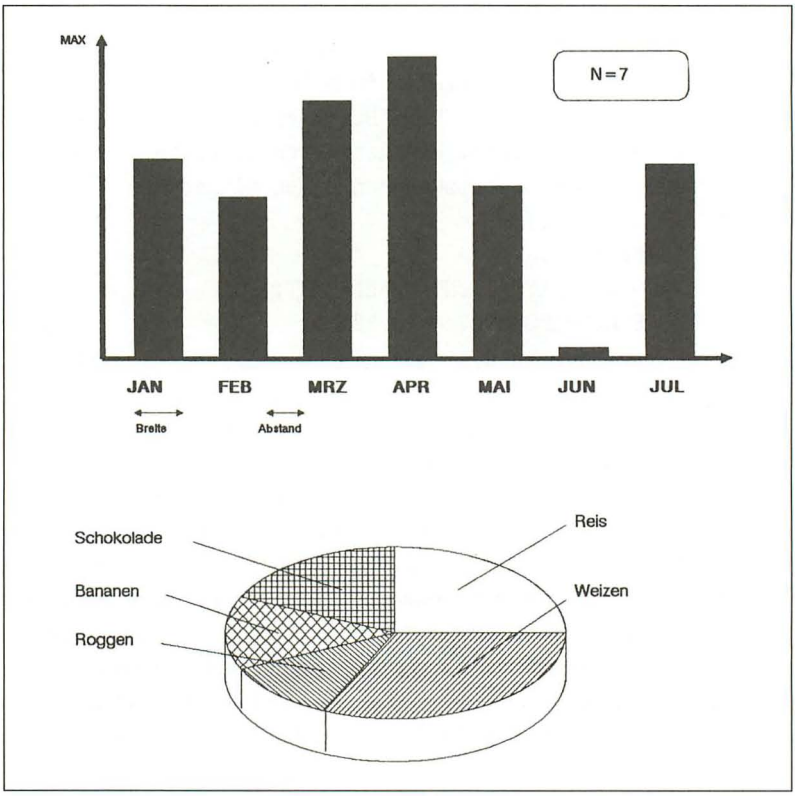


Bild 54: Balken- und Tortendiagramme

```
PROCEDURE BALKENDIAGRAMM (N       : INTEGER;
                          MAX     : REAL;
                          VAR B   : TWERTE;
                          VAR L   : TLEGENDE;
                          ORGX    : INTEGER;
                          ORGY    : INTEGER;
                          BREITE  : INTEGER;
                          ABSTAND: INTEGER);
(* Zeichne ein Balkendiagramm aus N Werten in B mit  *)
(* der Beschriftung L. Ausser den Balken wird noch   *)
(* eine X- und Y-Achse gezeichnet. ORGX und ORGY     *)
(* legen den Ursprung in Zeichenpositionen im 40*25  *)
(* Raster fest. Auch die Breite und der Abstand der  *)
(* Balken werden in Zeichenpositionen angegeben.     *)
```

Das Balkendiagramm soll eine feste Höhe von 150 Pixeln auf dem Bildschirm besitzen. Dabei definiert MAX den maximalen Wert, den ein Balken voller Größe besitzen müßte. Die Werte in B werden dementsprechend mit

einem Proportionalitätsfaktor multipliziert, um die Höhe jedes Balkens zu bestimmen. Mit ORGX und ORGY kann das Diagramm auf dem Bildschirm verschoben werden. Zur Anpassung der Balkenbreite an die darunterstehenden Texte werden die Parameter BREITE und ABSTAND in Zeichenpositionen angegeben.

```
PROCEDURE TORTE (N                          : INTEGER;
                 VAR T                      : TWERTE;
                 VAR L                      : TLEGENDE;
                 ORGX, ORGY, RADIUS, HOEHE: INTEGER)
(* Zeichne ein Tortendiagramm mit N Segmenten, die   *)
(* mit den Texten in L beschriftet werden. ORGX und  *)
(* ORGY legen den Mittelpunkt der Torte fest. HOEHE  *)
(* bestimmt die 'Dicke' der dreidimensionalen Torte  *)
```

Die Größe jedes Segmentes der Torte erhält man, indem man zunächst die Werte in T von 1 bis N aufaddiert. Dieser Wert entspricht einem Segment des Winkels 2*PI (6.2831853). Anschließend kann man durch einfache Dreisatzrechnung die Winkelgröße jedes Segmentes berechnen.

Wenn Sie ein dreidimensionales Diagramm zeichnen möchten, müssen Sie außerdem nur bei den »vorne« sichtbaren Segmenten einen Strich am »Rand« der Torte zeichnen. Besonders aufwendig ist die Plazierung der Texte zu den Segmenten. Dabei sollten Sie vom Mittelpunkt jedes Segmentes einen Strich zum zugehörigen Text ziehen. Dieser Text befindet sich bei Segmenten, deren Mittelpunktswinkel kleiner als PI (3.1415926) ist, auf der rechten Seite (sonst links von der Torte).

Soll die Textverteilung bei beliebigen Segmentgrößen »gut« aussehen, müssen Sie verhindern, daß sich die Texte nebeneinanderliegender Segmente überlappen. Wahrscheinlich ist diese Textverteilung die aufwendigste Teilaufgabe bei der Programmierung.

Bei beiden Diagrammtypen können Sie noch die Umrandungen mit dem Befehl PAINT ausfüllen. Jedoch müssen Sie den Startpunkt zum Ausmalen *innerhalb* der Umrandung wählen.

Bestimmung der Rechengenauigkeit für reelle Zahlen

3.
```
FUNCTION GENAUIGKEIT: REAL;
  VAR R: REAL;
BEGIN
  R:= 1.0;
  REPEAT
    R:= R * 0.5
  UNTIL R+1.0 <= 1.0;
  GENAUIGKEIT:= R;
END; (* GENAUIGKEIT *)
```

Welches Ergebnis liefert die obige Funktion? Sollte Ihnen die etwas eigenartige Bedingung am Ende der REPEAT-Anweisung Schwierigkeiten bereiten, sollten Sie Abschnitt 2.6 über die Darstellung reeller Zahlen im Computer zu Rate ziehen.

Die Türme von Hanoi

4. Das Standardbeispiel für die Anwendung rekursiver Prozeduren sind die *Türme von Hanoi*: Gegeben sind drei Stäbe und N Scheiben, die auf die Stäbe gesteckt werden können. Die Scheiben sind der Größe nach numeriert. Zu Beginn sind alle N Scheiben der Größe nach sortiert zu einem Turm auf dem ersten Stab gestapelt (siehe Bild 55).

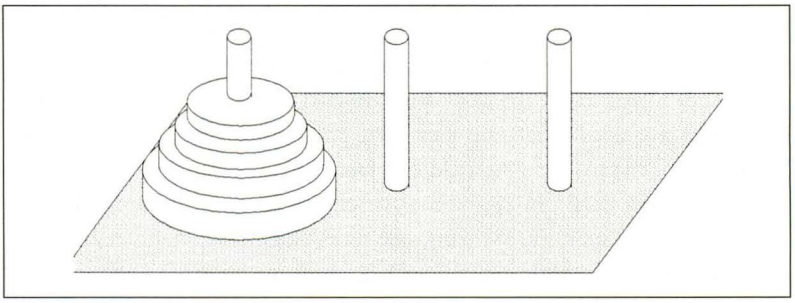


Bild 55: Die Türme von Hanoi

Die Aufgabe ist nun, den Turm in der gleichen Form auf Stab 3 aufzubauen. Dabei darf jedoch in jedem Schritt nur eine Scheibe von einem Stab zum anderen verlegt werden. Außerdem darf nie eine größere Scheibe auf einer kleineren liegen. Eine Lösung für 3 Scheiben sieht z.B. folgendermaßen aus:

Scheibe (1) von 1 nach 3
Scheibe (2) von 1 nach 2
Scheibe (1) von 3 nach 2
Scheibe (3) von 1 nach 3
Scheibe (1) von 2 nach 1
Scheibe (2) von 2 nach 3
Scheibe (1) von 1 nach 3

Schreiben Sie eine rekursive Prozedur, die einen Turm der Höhe N von 1 nach 3 verlegt und dabei den Lösungsweg wie oben angegeben protokolliert. Die Idee zu einer rekursiven Lösung besteht darin, daß zuerst ein Turm der Höhe N–1 von 1 auf den Hilfsstab 2 gebracht werden muß, um die Scheibe N von 1 nach 3 zu verlegen. Anschließend kann man den Turm der Höhe N–1 von 2 nach 3 bewegen und hat somit die Aufgabe gelöst.

Der Algorithmus läßt sich mit sehr wenigen Anweisungen **ohne** die Verwendung eines Arrays programmieren. Erst wenn Sie die rekursive Lösung gefunden haben, sollten Sie sich überlegen, wie man das Verlegen der Scheiben grafisch darstellen könnte. Entweder verwenden Sie dann ein

Array, das die momentane Anordnung aller Scheiben speichert, oder Sie übergeben alternativ der rekursiven Prozedur zusätzlich die Höhe jedes Turmes.

5. Haben Sie bereits in BASIC programmiert, sollten Sie einige Ihrer BASIC-Programme in Pascal neu formulieren. Inzwischen ist Ihr Wissen in Pascal so weit gediehen, daß Sie zu jeder Kontroll- und Datenstruktur in BASIC das Gegenstück in Pascal kennen. Wahrscheinlich werden Sie erkennen, daß Sie durch die Blockstruktur von Pascal gezwungen sind, den Programmablauf völlig neu zu organisieren.

2.12 Skalare Typen und ihre Operationen

Skalare Typen für problemorientierte Datenstrukturen

Dieser Abschnitt stellt eine Fortsetzung von Kapitel 2.6 über die Typen der Sprache Pascal dar. Es werden Methoden vorgestellt, um neue skalare Typen zu deklarieren. Die so definierten Typen erlauben es, im Computer ein möglichst exaktes Modell der zu bearbeitenden Daten zu bilden. Auch wenn auf der Ebene der Maschinensprache alle Daten als binäre Zahlenfolgen dargestellt werden, sollte der Programmierer problemorientierte Objekte definieren und manipulieren können.

2.12.1 Aufzählungstypen

Im Rahmen einer Typdeklaration kann man eine Menge von Werten aufzählen und zu einem Typ zusammenfassen:

```
TYPE WOTAG     =(MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,
                FREITAG, SAMSTAG, SONNTAG);
     FAMSTAND =(LEDIG, VERHEIRATET, GETRENNT, GESCHIEDEN,
                VERWITWET);
     FRUCHT    =(APFEL, BIRNE, ORANGE);
VAR  HEUTE, MORGEN  : WOTAG;
     LIEBLINGSFRUCHT: FRUCHT;
     SPEISEPLAN     : ARRAY [WOTAG] OF FRUCHT;
     STATUS         : FAMSTAND;
```

Bild 56: Aufzählungstypen

Deklaration eines Aufzählungstyps

Die Variablen HEUTE und MORGEN können nur die Werte Montag bis Sonntag annehmen. WOTAG, FAMSTAND und FRUCHT bezeichnet man als **Aufzählungstypen**. Die Namen in Klammern sind Konstanten des jeweiligen Aufzählungstyps. MONTAG ist also eine Konstante vom Typ WOTAG.

Operationen mit Aufzählungstypen

Zwar kann man mit Werten eines Aufzählungstyps nicht »rechnen«, jedoch sind Zuweisungen und Vergleiche zwischen Variablen eines Aufzählungstyps möglich:

```
HEUTE:= SAMSTAG;
IF HEUTE = MORGEN THEN WRITELN('???');
CASE SPEISEPLAN[SAMSTAG] OF
 APFEL : WRITELN('Apfel');
 BIRNE : WRITELN('Birne');
 ORANGE: WRITELN('Orange');
END;
SPEISEPLAN[HEUTE]:= SPEISEPLAN[MORGEN];
```

Wegen der Typbindung in Pascal sind folgende Operationen fehlerhaft:

```
HEUTE:= LIEBLINGSFRUCHT;
SPEISEPLAN[3]:= APFEL;
IF HEUTE = LEDIG THEN ...
```

Ordnung der Konstanten

Durch die Reihenfolge der Konstanten bei der Typdeklaration wird eine Ordnung definiert, so daß auch Vergleiche mit »<« und »>« sinnvoll sind:

```
ORD(MONTAG)   = 0
ORD(DIENSTAG) = 1
ORD(SONNTAG)  = 6

MONTAG<DIENSTAG        MITTWOCH> MONTAG
LEDIG < VERWITWET
IF HEUTE<=FREITAG THEN WRITELN('Heute ist ein Werktag')
IF LIEBLINGSFRUCHT >APFEL THEN WRITELN('Birne oder Orange')
```

SUCC und PRED

Die Standardfunktionen SUCC und PRED liefern zu jedem skalaren Typ den Nachfolger und Vorgänger im Wertebereich.

```
SUCC(DIENSTAG)     = MITTWOCH
PRED(SONNTAG)      = SAMSTAG
SUCC(APFEL)        = BIRNE
PRED(VERHEIRATET)  = LEDIG
```

aber auch

```
SUCC(1)            = 2
SUCC(-2)           = -1
SUCC(FALSE)        = TRUE
SUCC('A')          = 'B'
```

Dementsprechend läßt sich die FOR-Anweisung auch mit einer Laufvariablen eines Aufzählungstyps verwenden:

```
FOR HEUTE:= MONTAG TO FREITAG DO
  FOR MORGEN:= SUCC(HEUTE) TO SONNTAG DO
    WRITELN(ORD(HEUTE):4, ORD(MORGEN):4);
```

Im Computer existiert zur Laufzeit eines Objektprogrammes nur die kompakte codierte Darstellung der Aufzählungswerte durch ihre Ordinalwerte. Deshalb kann man die Namen der Werte des Aufzählungstyps nicht direkt ausgeben:

```
WRITE('Heute ist ', HEUTE)
```

Falsch!

Statt dessen muß man die Ausgabe explizit programmieren:

Ausgabe von Werten eines Aufzählungstyps

```
PROGRAM TAGE (OUTPUT);
  TYPE WOTAG = (MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,
                FREITAG, SAMSTAG, SONNTAG);
  VAR  HEUTE : WOTAG;
PROCEDURE WRITEWOTAG (TAG: WOTAG);
BEGIN
  CASE TAG OF
    MONTAG   : WRITE('Montag');
    DIENSTAG : WRITE('Dienstag');
    ...
    SONNTAG  : WRITE('Sonntag');
  END; (* CASE *)
END; (* WRITEWOTAG *)
BEGIN
  FOR HEUTE:= SONNTAG DOWNTO MONTAG DO
    BEGIN WRITEWOTAG(HEUTE); WRITE(','); END;
END.
```

Bild 57: Ausgabe eines Aufzählungstyps

Aufzählungstypen erhöhen die Lesbarkeit eines Programmes enorm und sollten wo immer möglich verwendet werden. Ein etwas spezielleres Beispiel zeigt die Verwendung eines Aufzählungstyps als Indextyp: Im Programm soll die Anzahl der Bürger jedes Familienstandes gezählt werden.

Aufzählungstypen als Indextypen für Arrays

Die Idee besteht darin, ein Array von Zählern zu deklarieren, das direkt durch Werte des Typs FAMSTAND indiziert wird. Damit kann man in einer Schleife, die alle Bürger erfaßt, mit einer einzigen Zuweisung den jeweiligen Zähler erhöhen (siehe Bild 58):

```
PROGRAM VOLKSZAEHLUNG (INPUT, OUTPUT);
(* Dieses Programm skizziert die Verwendung von Auf- *)
(* zählungstypen als Indextypen.                     *)
  CONST ANZAHLBUERGER = 300;
  TYPE FAMSTAND = (LEDIG, VERHEIRATET, GETRENNT, GESCHIEDEN,
                   VERWITWET);
  VAR  STAMMBUCH: ARRAY [1..ANZAHLBUERGER] OF FAMSTAND;
       STATISTIK: ARRAY [FAMSTAND] OF INTEGER;
       STATUS   : FAMSTAND;
       BUERGER  : INTEGER;
```

Bild 58: Aufzählungstypen als Indizes (Fortsetzung nächste Seite) ►

```
  PROCEDURE STAMMBUCHBELEGEN;
   (* Für jeden Bürger wird in dieser Prozedur der   *)
   (* Familienstand im Stammbuch zufällig belegt.    *)
    VAR I: INTEGER;
  BEGIN
    FOR I:= 1 TO ANZAHLBUERGER DO
      STAMMBUCH[I]:= FAMSTAND (INT(RANDOM(0) *
                                (ORD(VERWITWET)+1)));
  END; (* STAMMBUCHBELEGEN *)

 BEGIN
  STAMMBUCHBELEGEN;
  (* zunächst alle Zähler zurücksetzen: *)
  FOR STATUS:= LEDIG TO VERWITWET DO
    STATISTIK[STATUS]:= 0;
  (* jetzt Stammbuch durchlaufen und zählen *)
  FOR BUERGER:= 1 TO ANZAHLBUERGER DO
    BEGIN
      STATUS:= STAMMBUCH[BUERGER];
      STATISTIK[STATUS]:= STATISTIK[STATUS] + 1;
    END;
  (* Ergebnisse drucken: *)
  FOR STATUS:= LEDIG TO VERWITWET DO
    WRITELN( ORD(STATUS):4, STATISTIK[STATUS]:8);
END.
```

Bild 58: Aufzählungstypen als Indizes (Schluß)

Zufällige Werte für Aufzählungstypen

Die Prozedur STAMMBUCHBELEGEN verwendet das Gegenstück zur Prozedur ORD, das jedoch nicht im *report* definiert ist: Um einen zufälligen Familienstand zu definieren, wird die Funktion RANDOM(0) verwendet. Sie liefert eine reelle Zahl X, die aus dem Intervall $0 <= X < 1$ stammt. Dieser Wert wird durch die Multiplikation mit der Zahl

`(ORD(VERWITWET)+1) = 5`

auf das Intervall 0 bis 4.999999 abgebildet. Nachdem mit der Funktion INT die Nachkommastellen abgeschnitten wurden, wird die so berechnete INTEGER-Zahl zwischen 0 und 4 mit dem Typnamen FAMSTAND in den Typ FAMSTAND konvertiert. Es gilt nämlich:

Explizite Typbindung in Pascal 2.0

```
FAMSTAND(0) = LEDIG
FAMSTAND(1) = VERHEIRATET
...
FAMSTAND(4) = VERWITWET
```

```
WOTAG(0)     = MONTAG
WOTAG(1)     = DIENSTAG
FRUCHT(2)    = ORANGE
```

2.12.2 Unterbereichstypen

Gibt es in einem Programm Variablen, die nur Werte aus einem Teilbereich des Wertebereichs eines Typs annehmen (oder annehmen sollen), so läßt sich diese Information bei der Deklaration einer Variablen angeben.

```
CONST N=3; M=4;
TYPE  ZEILENINDEX  = 1..N;
      SPALTENINDEX = 1..M;
      DATEITYP     = (SEQ, INDSEQ, REL, ERASED);
      XKOORDINATE  = 0..319;
      YKOORDINATE  = 0..199;
VAR   M: ARRAY[ZEILENINDEX, SPALTENINDEX] OF REAL;
      I, I1: ZEILENINDEX;
      J, J1: SPALTENINDEX;
      B, BUCHSTABE:  'A'..'Z';
      ZIFFER       :  '0'..'9';
      X,CENTERX    : XKOORDINATE;
      Y,CENTERY    : YKOORDINATE;
      ARBEITSDATEI: SEQ..REL;
```

Bild 59: Unterbereichstypen

Deklaration eines Unterbereichstyps

Bei einer Typ- oder Variablendeklaration kann man einen sogenannten Unterbereichstyp deklarieren, indem man getrennt durch das Auslassungssymbol »..« die untere und obere Grenze innerhalb eines anderen Typs nennt. Der so definierte Typ kann wie jeder skalare Typ in Variablendeklarationen oder zusammengesetzten Typdeklarationen verwendet werden.

Typkompatibilität

Werte eines Unterbereichstyps können an jeder Stelle verwendet werden, an der auch ein Wert des zugrundeliegenden Basistyps zulässig ist. Außerdem sind mit Variablen eines Unterbereichstyps alle Operationen des Basistyps möglich:

```
I:= I1 * 2;
A[I,J]:= I + J * 4 - 5;
B:= CHR(68);
ARBEITSDATEI:= SEQ;
X:= X DIV 2;
```

Sicherheit und Lesbarkeit durch Unterbereichstypen

Obwohl die Einführung von Unterbereichstypen etwas mehr (Schreib-) Aufwand bei der Programmierung erfordert, ermöglicht sie eine zusätzliche Sicherheit vor unzulässigen Zuweisungen und erlaubt so eine einfachere Fehlersuche. Mindestens ebenso wichtig ist der erhöhte Dokumentationswert einer Variablendeklaration:

- Die ARBEITSDATEI kann nie gelöscht (ERASED) sein.
- Die Werte der Variablen I und J bilden offensichtlich Indizes in der Matrix M.
- Der Variablen B dürfen nur Großbuchstaben zugewiesen werden.
- X, CENTERX und Y, CENTERY stellen sichtbare Koordinaten auf dem Grafikbildschirm dar und dürfen daher den Bereich 0..319 und 0..199 nicht verlassen.

Bereichsüberprüfung zur Programmlaufzeit

Wählen Sie bei der Kompilation die Option *range-check* (Bereichsüberprüfung) des Compilers, indem Sie den *aktiven Kommentar*

```
(*$R+    *)
```

im Programm verwenden, so wird das Programm nachfolgend um Codesequenzen erweitert, die bei der Laufzeit auch die Einhaltung der Intervallgrenzen bei Unterbereichstypen prüfen. Dabei müssen nur die Zuweisungen an Variablen eines Unterbereichs geprüft werden:

```
I:= I1 *2;
READLN(BUCHSTABE);
ZIFFER:= SUCC(ZIFFER);
FOR J:= -3 TO 3 DO WRITE(J);
M[I+1,J+1]:= 30;
```

Verwendet man jedoch durchgängig im gesamten Programm Unterbereichstypen, so **sinkt** die Anzahl der Stellen, an denen Code zur Überprüfung erzeugt werden muß. Bei Zuweisungen der folgenden Art sind nämlich keine Überprüfungen erforderlich:

```
J:= J1;
I:= I1;
B:= BUCHSTABE;
M[I,J]:= M[I1,J1]
```

Fehler bei Unterbereichstypen

Wurde die Bereichsüberprüfung eingeschaltet, wird bei einer illegalen Zuweisung im Objektprogramm eine Fehlermeldung erzeugt:

```
ZIFFER:= SUCC('9');
VALUE OUT OF BOUNDS: 58  48  57
```

Dies bedeutet, daß der Nachfolger des Zeichens »9« mit dem Ordinalwert 58 nicht im Bereich der Ziffern »0« bis »9« mit den Ordinalwerten 48 und 57 liegt. Also:

Fehlerhafter Wert	ORD(SUCC('9'))	=	58
untere Bereichsgrenze	ORD('0')	=	48
obere Bereichsgrenze	ORD('9')	=	57

Generell werden bei dieser Fehlermeldung nur die Ordinalwerte angegeben, da z.B. für Aufzählungstypen im Objektprogramm keine Namen vorhanden sind.

Zumindest in der Testphase ist die Option *range check* (siehe Abschnitt 4.4.6.1) sehr zu empfehlen, um Indizierungsfehler und Bereichsüberschreitungen zu entdecken. Die Ausführungsgeschwindigkeit und Codegröße steigt durch diese Option nur geringfügig an. Im endgültigen Programm kann man die Option wieder ausschalten, um optimalen Code zu erhalten.

Speicherplatzeinsparung durch Unterbereichstypen

Abschließend sollte erwähnt werden, daß in einzelnen Fällen die Verwendung von Unterbereichstypen den Speicherplatzbedarf für Variablen verringern kann. Um das Alter von 100 Personen zu speichern, sind folgende zwei Arraydeklarationen möglich:

```
VAR ALTER1: ARRAY[1..100] OF 0..130;
    ALTER2: ARRAY[1..100] OF INTEGER;
```

Während die Variable ALTER1 100 Byte belegt, werden durch die Variable ALTER2 200 Byte belegt. Gerade bei großen Datenstrukturen sind solche Optimierungsmöglichkeiten zu beachten. In Pascal 2.0 existiert bereits der vordefinierte Unterbereichstyp BYTE, der genau ein Byte Speicherplatz belegt:

```
TYPE BYTE =.0..255;
```

Aufgaben

1. Untersuchen Sie alle bisher im Text angegebenen Beispielprogramme. Prüfen Sie, ob in den Variablendeklarationen die Möglichkeit besteht, Unterbereichstypen zu verwenden. Prädestiniert für solche Verbesserungen sind Variablen, die zur Indizierung verwendet werden. (Diese Variablen heißen meist I oder J.)
   Kompilieren Sie ein Beispielprogramm mit der Range-check-Option, und prüfen Sie die Reaktion auf Bereichsüberschreitungen!
2. Modifizieren Sie das Programm QUICKSORT aus Bild 52 durch die Einführung von Typnamen (INDEX und ITEM) im Hauptprogramm, so daß beliebige Indexbereiche und Elementtypen sortiert werden können. Prüfen Sie die Richtigkeit der Änderungen, indem Sie folgendes Array sortieren:

```
TYPE INDEX = 10..20;
     ITEM  = CHAR;
VAR  A : ARRAY [INDEX] OF ITEM;
```

2.13 Mengentypen

Deklaration Mengen skalarer Typen

Neben den Arrays gibt es in Pascal noch weitere zusammengesetzte Typen, die jeweils für spezielle Anwendungsgebiete besonders geeignet sind. Zu jedem skalaren Typ T läßt sich ein Typ SET OF T deklarieren, der als Werte alle möglichen **Mengen** von Werten des Typs T annimmt:

```
TYPE FRUCHT      = (APFEL, BIRNE, ORANGE);
     OBST        = SET OF FRUCHT;
     FAMSTAND    = (LEDIG, VERHEIRATET, GETRENNT, GESCHIEDEN,
                    VERWITWET);
VAR  LIEFERBAR, AUSVERKAUFT: OBST;
     ZAHLENMENGE = SET OF 1..30;
     KOMMANDOS   = SET OF 'A' .. 'E';
     STATISTIK2:   SET OF FAMSTAND;
```

Die Variablen LIEFERBAR und AUSVERKAUFT können also die folgenden Werte annehmen:

```
[ ]
[APFEL]
[BIRNE]
[ORANGE]
[APFEL, BIRNE]
[APFEL, ORANGE]
[BIRNE, ORANGE]
[APFEL, BIRNE, ORANGE]
```

Definition der Mengenoperationen in Pascal

Dies sind alle möglichen Mengen, die aus den drei Früchten gebildet werden können (2 hoch 3 Möglichkeiten). Um die Operationen mit Mengen in Pascal zu verstehen, müssen Sie sich nur an den Mengenlehreunterricht erinnern und die in der Mathematik üblichen geschweiften Mengenklammern (»{« »}«) durch die eckigen Klammern (»[« »]«) in Pascal ersetzen. Praktisch alle Operationen der Mengenlehre sind auch in Pascal verfügbar. Sind A und B Mengenausdrücke des gleichen Typs, so sind folgende Operationen möglich:

A + B	bildet die Vereinigungsmenge von A und B, das sind alle Elemente, die in A oder B enthalten sind.
A * B	bildet die Schnittmenge von A und B, das sind alle Elemente, die gleichzeitig in A und B enthalten sind.
A – B	bildet die Differenzmenge von A und B, das sind alle Elemente, die in A und nicht in B enthalten sind.
A = B	testet die Mengen A und B auf Gleichheit. Das heißt, jedes Element von A ist Element von B und umgekehrt.
A <>B	liefert das Ergebnis NOT (A=B).
A <= B	prüft, ob A Teilmenge von B ist.

A >= B	prüft, ob A Obermenge von B ist.
a in B	prüft, ob das Element a in der Menge B enthalten ist.
[ ]	bildet die leere Menge.
[a,b]	bildet die Menge, die aus den Elementen a und b besteht.
[a..b]	bildet die Menge, die alle Werte zwischen a und b (einschließlich) beinhaltet. Ist a>b, so bezeichnet [a..b] die leere Menge [].

Wichtig ist die Tatsache, daß A und B den gleichen Elementtyp besitzen:

```
[LEDIG, VERHEIRATET] + [APFEL..BIRNE]
```

ist offenbar ein sinnloser Ausdruck. Wie in der Mathematik ist [APFEL] nicht gleich APFEL. Das erste ist eine einelementige Menge (vom Typ OBST) und das andere ein Wert des Aufzählungstyps FRUCHT. Außerdem enthält eine Menge kein Element doppelt.

Anwendungsbeispiel für Mengenoperationen

Allgemein gesprochen werden Mengen überall dort verwendet, wo Werte eines Typs in ungeordnete Gruppen mit gewissen Eigenschaften aufgeteilt werden sollen. Um zum Beispiel in einem Gemüseladen Buch über die lieferbaren Früchte zu führen, eignen sich die Mengenvariablen LIEFERBAR und AUSVERKAUFT.

```
LIEFERBAR:= [APFEL, BIRNE, ORANGE];
AUSVERKAUFT:= [];
LIEFERBAR:= LIEFERBAR + [BIRNE];
AUSVERKAUFT:= AUSVERKAUFT - [BIRNE];
LIEFERBAR:= LIEFERBAR * [BIRNE..ORANGE];
IF [APFEL,ORANGE]<= LIEFERBAR THEN
  WRITELN('Apfel und Orange lieferbar');
IF LIEFERBAR * AUSVERKAUFT <>[] THEN
  WRITELN('Fehler in der Buchführung');
IF APFEL IN AUSVERKAUFT THEN
  LIEFERBAR:= LIEFERBAR - AUSVERKAUFT
```

Sie sollten sich die Mühe machen, die Bedeutung jeder einzelnen Anweisung in Worte zu fassen, um die teilweise recht komplexen Operationen zu verstehen. Mengenoperationen mit Mengen des Typs SET OF CHAR werden in vielen Programmen zur Vereinfachung von Abfragen benutzt:

Mengenoperationen zur Programmsteuerung

```
REPEAT READ(CH) UNTIL CH IN ['J','Y','N'];
IF CH IN ['0'..'9'] THEN ...
IF CH IN ['0'..'9','A'..'Z'] THEN ...
```

Grundsätzlich beschränkt jeder Computer die Größe einer Menge. Als Grundtyp für Mengen scheidet deshalb neben dem Typ REAL auch der Typ INTEGER aus. Zum Beispiel gibt es bereits für eine Variable des (zulässigen) Typs SET OF 1..30 genau 2 hoch 30 = 10737341824 verschiedene Werte!

Begrenzung der Mengengröße

Pascal 2.0 erlaubt nur Mengen von Typen, die nicht mehr als 96 Werte annehmen können. Durch diese Grenze können auch Mengen von Zeichen (SET OF

CHAR) dargestellt werden, die jedoch **keine** Grafikzeichen (respektive Großbuchstaben) enthalten dürfen, für die gilt:
ORD(CH) > =96

Demonstrationsprogramm für alle Operationen

Das Programm in Bild 60 zeigt die Wirkung der Mengenoperationen am Beispiel des Typs SET OF CHAR. Nach den obigen Erläuterungen erübrigt sich eine weitere Diskussion des Programm-Listings:

```
PROGRAM MENGENLEHRE (INPUT,OUTPUT);
  TYPE MENGE = SET OF CHAR;
  VAR  A,B : MENGE;
       E   : CHAR;

  PROCEDURE EINGABE(VAR M: MENGE);
    VAR CH: CHAR;
  BEGIN
    M:= []; READ(CH);
    WHILE NOT EOLN DO
      BEGIN M:= M + [CH]; READ(CH); END;
  END; (* EINGABE *)

  PROCEDURE AUSGABE (M: MENGE);
    VAR CH: CHAR;
    BEGIN
      FOR CH:= ' ' TO CHR(95) DO
        IF CH IN M THEN WRITE(CH:3);
      WRITELN;
    END; (* AUSGABE *)

BEGIN
  WRITE('Menge A:'); EINGABE(A);
  WRITE('Menge B:'); EINGABE(B);
  WRITE('Menge A:'); AUSGABE(A);
  WRITE('Menge B:'); AUSGABE(B);
  WRITE('A + B  :'); AUSGABE(A+B);
  WRITE('A * B  :'); AUSGABE(A*B);
  WRITE('A - B  :'); AUSGABE(A-B);
  WRITE('A = B  :'); WRITELN(A=B);
  WRITE('A<= B :'); WRITELN(A<=B);
END.
```

Bild 60: Mengenoperationen

Sieb des Eratosthenes

Wegen der sehr kompakten Speicherung der Werte eignen sich Mengen auch für die folgende Anwendung. Es sollen die Primzahlen von 1 bis 10000 gedruckt werden. Das einfachste Verfahren zur Bestimmung von Primzahlen wird durch das *Sieb des Eratosthenes* beschrieben:
Man beginnt mit einer *Tabelle* aller Zahlen im Intervall 1 bis 10000. Nun streicht man nacheinander alle Vielfachen der Zahlen 2, 3, 5, 7, 11 etc. Am Schluß bleiben also nur die Primzahlen in der Tabelle stehen, da sie durch keine der anderen Zahlen teilbar sind.

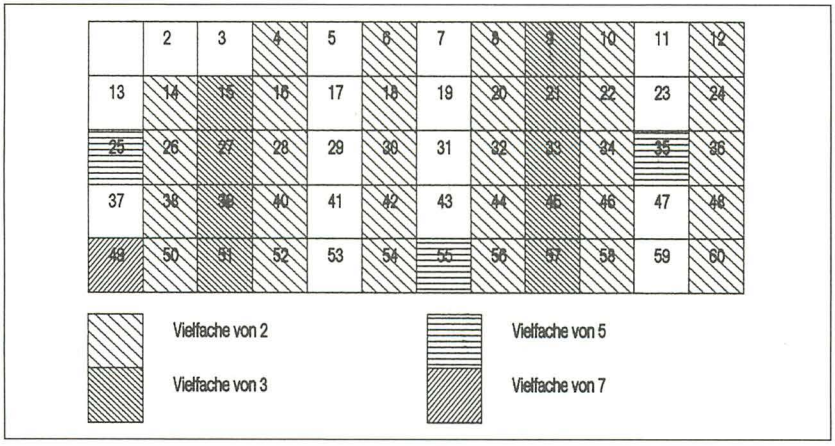


Bild 61: Sieb des Eratosthenes

Bild 61 zeigt den Anfang der Tabelle, wobei jeweils die Vielfachen von 2, 3, 5 und 7 gestrichen wurden. Man könnte also das Streichen der Vielfachen folgendermaßen in Pascal programmieren:

```
PROGRAM SIEB1 (INPUT, OUTPUT);
(* Dieses Programm ist nicht korrekt, da die Größe einer *)
(* Menge 96 Elemente nicht überschreiten darf.           *)
  CONST MAX = 10000;
  VAR TEILBAR: SET OF 0..MAX;
      P, Z, I: INTEGER;
BEGIN
  TEILBAR:=[]; (* noch ist keine Zahl gestrichen worden *)
  P:=1;
  REPEAT
    (* suche nächste Primzahl als Teiler *)
    REPEAT
      P:=P+1
```

Bild 62: Programm SIEB1 (Fortsetzung nächste Seite)

```
    UNTIL NOT( P IN TEILBAR);
    (* streiche Vielfache von P *)
    Z:= P*P;
    WHILE Z<=MAX DO
      BEGIN
        TEILBAR:= TEILBAR + [Z];
        Z:= Z + P;
      END;
  UNTIL P*P>MAX;
  (* Drucke die übriggebliebenen Primzahlen: *)
  FOR I:= 2 TO MAX DO
    IF NOT(I IN TEILBAR) THEN WRITE(I:6);
  WRITELN;
END.
```

Bild 62: Programm SIEB1 (Schluß)

Primzahlberechnung in Pascal 2.0

Wie bereits erwähnt, dürfen Mengen in *Pascal 2.0* maximal 96 Elemente beinhalten. Daher wird die Menge TEILBAR aus Bild 62 durch ein Array von Mengen mit jeweils 96 Elementen realisiert. Die erste Menge enthält also die Zahlen von 0 bis 95, die zweite die Zahlen zwischen 96 und 191 etc. Damit erhält man das korrekte Programm in Bild 63:

```
PROGRAM SIEB2 (INPUT, OUTPUT);
(* Dieses Programm bestimmt alle Primzahlen im Bereich von *)
(* 1 bis MAX. Dabei simuliert das Array TEILBAR eine Menge *)
(* mit MAX Elementen. Die Darstellung erfolgt durch MAX96  *)
(* Mengen mit jeweils SETSIZE Elementen                    *)
      CONST MAX     = 10000;
            SETSIZE  = 96;        (* Für Pascal 2.0        *)
            MAX96    = 105;       (* = MAX DIV SETSIZE + 1 *)
     VAR TEILBAR: ARRAY[0..MAX96] OF SET OF 0..95;
            P, Z, I: INTEGER;

 FUNCTION PRIM(Z:INTEGER): BOOLEAN;
 (* Prüfe, ob die Zahl Z prim ist, d.h. Z ist nicht in    *)
 (* der Menge der teilbaren Zahlen                        *)
 BEGIN
    PRIM:= NOT((Z MOD SETSIZE) IN TEILBAR[Z DIV SETSIZE])
 END; (* PRIM *)
```

Bild 63: Programm SIEB2

```
BEGIN
  FOR I:= 0 TO MAX96 DO
    TEILBAR[I]:=[]; (* noch wurde keine Zahl gestrichen *)
  P:=1;
  REPEAT
    (* suche nächste Primzahl als Teiler *)
    REPEAT
      P:=P+1
    UNTIL PRIM(P);
    (* streiche Vielfache von P *)
    WRITELN('Streiche Vielfache von', P:4);
    Z:= P*P;
    WHILE Z<=MAX DO
      BEGIN
        I:= Z DIV SETSIZE;
        TEILBAR[I]:= TEILBAR[I] + [Z MOD SETSIZE];
        Z:= Z + P;
      END;
  UNTIL P*P>MAX;
  (* Drucke die übriggebliebenen Primzahlen: *)
  FOR I:= 2 TO MAX DO
    IF PRIM(I) THEN WRITE(I:6);
  WRITELN
END.
```

Bild 63: Programm SIEB2 (Schluß)

Aufgaben

1. Schreiben Sie eine Funktion, die die Anzahl der Elemente in einer Menge als Ergebnis liefert.
2. Geben Sie Mengenausdrücke an, durch welche die in den *Venn-Diagrammen* (Bild 64, Seite 148) schraffierten Mengen berechnet werden.

```
TYPE MENGE = SET OF 0 .. 95;
VAR  A, B: MENGE;
A:= [1,3,5,7];
B:= [2,4,6,7,10];
```

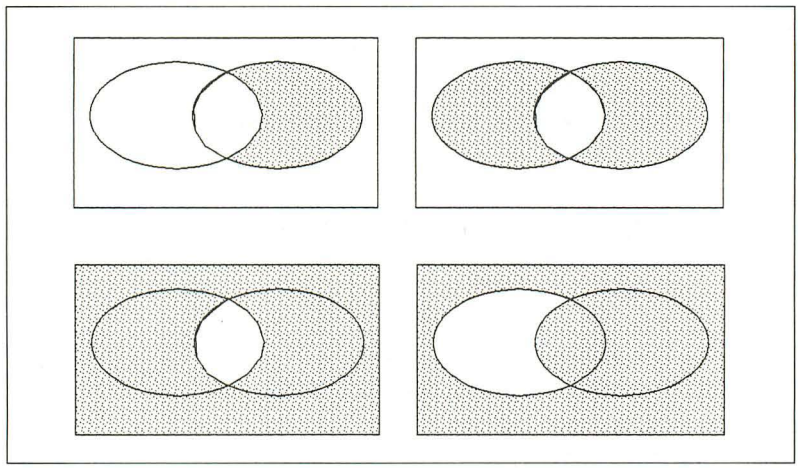


Bild 64: Venn-Diagramme

3. Schreiben Sie ein Programm, das die ersten 500 Primzahlen in einem Array A speichert. Im Gegensatz zum Programm SIEB2 prüfen Sie hierbei für jede Zahl, ob sie durch eine der bereits gespeicherten kleineren Primzahlen teilbar ist. Sie können den Algorithmus verbessern, indem Sie von Anfang an die Vielfachen von 2 und 3 ausschließen. Vergleichen Sie die Programmlaufzeiten!

2.14 Der Datentyp Record

Records fassen Objekte verschiedener Typen zusammen

In einem Array und einer Menge werden Elemente eines einzigen Typs zu einer Datenstruktur zusammengefaßt. Um Werte verschiedener Typen zu verbinden, benutzt man in Pascal Record-Variablen:

Deklaration

```
TYPE  KENNZEICHEN = RECORD
                      KREIS: STRING[3];
                      B    : STRING[2];
                      NR   : 1..9999;
                    END;
      ADRESSE     = RECORD
                      NAME,VORNAME: STRING[20];
                      ORT,STRASSE : STRING[15];
                      HAUSNR, PLZ : INTEGER;
                    END;
      KRAFTFAHRZEUGSCHEIN =
                    RECORD
                      WAGEN            : KENNZEICHEN;
```

Bild 65: Record-Typen

```
                    WOHNORT,STANDORT : ADRESSE;
                    LEISTUNG         : INTEGER;
                    ABGEMELDET       : BOOLEAN;
                  END;

VAR  HALTER            : ADRESSE;
     AUTO1, AUTO2      : KENNZEICHEN;
     SCHEIN1, SCHEIN2: KRAFTFAHRZEUGSCHEIN;
```

Bild 65: Record-Typen (Schluß)

Dieses etwas ausführlichere Beispiel zeigt, wie man Attribute eines realen Objektes (Fahrzeugkennzeichen, Adresse, Fahrzeugschein) in einer Variablen vom Typ Record speichert: Ein Kennzeichen besteht aus einem dreistelligen Kürzel für den Kreis, zwei Buchstaben und einer Nummer. Im Fahrzeugschein werden für ein Fahrzeug der Halter und der Standort des Fahrzeugs eingetragen.

Records als »Karteikasten«

Man kann sich einen Record wie eine »Karteikarte« mit vordefinierten Feldern vorstellen, die nur Werte bestimmter Typen beinhalten dürfen. Die in Bild 65 definierten Typen bezeichnet man auch als Verbundtypen. Auf die **Felder** einer Record-Variablen greift man durch Nennung des Variablennamens und des Feldnamens, getrennt durch einen Punkt, zu:

Zugriff auf Record-Felder

```
AUTO1.KREIS:= 'M';
AUTO2.KREIS:= 'HH';
SCHEIN1.WOHNORT.ORT:='NEW YORK';
SCHEIN2.STANDORT.ORT:= HALTER.ORT;
IF SCHEIN1.LEISTUNG < 28 THEN ...
```

Der Record-Typ KRAFTFAHRZEUGSCHEIN beinhaltet seinerseits die Record-Typen KENNZEICHEN und ADRESSE als Felder. Daher kann man auch über mehrere Stufen auf Record-Felder zugreifen:

```
SCHEIN1.WAGEN.KREIS:= AUTO1.KREIS;
SCHEIN2.STANDORT.ORT:= SCHEIN2.WOHNORT.ORT;
```

In der ersten Zuweisung wird also der Inhalt des Feldes KREIS im Kennzeichen AUTO1 in dem Feld KREIS des Feldes WAGEN in der Record-Variablen SCHEIN1 gespeichert. Wirklich nützlich wird das Konzept der Records in Pascal durch die Tatsache, daß man Zuweisungen zwischen kompletten Records des gleichen Typs vornehmen kann:

```
AUTO1:= AUTO2;
SCHEIN1.WOHNORT:= HALTER;
SCHEIN1:= SCHEIN2;
```

Block-Zuweisungen zwischen Records

Durch solche *Block-Zuweisungen* werden alle Felder eines Records kopiert. Die Feldnamen (Selektoren), wie ORT, B und NR, sind Namen, deren Sichtbarkeit auf den Record ihrer Deklaration beschränkt ist. Deshalb könnte man also durchaus ohne Namenskonflikte eine Variable KREIS deklarieren, da Feldnamen vom Compiler an dem vorausgehenden Punkt erkannt werden können.

WITH-Anweisung zur Vereinfachung von Record-Operationen

Eine ähnliche Bedeutung wie die FOR-Anweisung für Arrays besitzt die sogenannte **WITH-Anweisung** (Inspektionsanweisung) für Variablen vom Typ Record. Sie vereinfacht Ausdrücke, die mit vielen Feldern eines Records arbeiten.

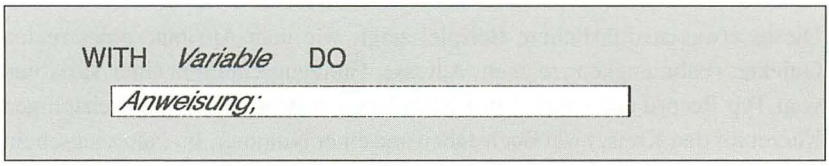


Bild 66: Struktur der WITH-Anweisung

```
WITH SCHEIN1 DO
  BEGIN
    WITH WAGEN DO
      BEGIN KREIS:='MTK'; B:='M'; NR:=939 END;
    WITH WOHNORT DO
      BEGIN
        NAME   := 'MUELLER THURGAU';
        VORNAME:= 'HANS PETER';
        PLZ    := 6232;
        HAUSNR := 4;
      END;
    STANDORT:= WOHNORT; LEISTUNG:=45;
    ABGEMELDET:= FALSE;
  END;
```

Geschachtelte WITH-Anweisungen

In der Anweisung, die nach dem Wortsymbol DO der WITH-Anweisung folgt, ist also die Angabe *Variablenbezeichner* vor dem Feldnamen nicht erforderlich. Geschachtelte WITH-Anweisungen beziehen sich aber immer auf dieselbe Record-Variable. Die folgende Schachtelung ist also verboten, da der Record AUTO1 nicht zum Record SCHEIN1 gehört:

Falsch!

```
WITH SCHEIN1 DO
  BEGIN
    WOHNORT:= HALTER;
    WITH AUTO1 DO
      B:=WAGEN.B;
  END;
```

Bei vielen Compilern (auch *Pascal 2.0*) bringt die WITH-Anweisung außerdem noch Geschwindigkeitsvorteile, da alle Operationen zum Zugriff auf die Record-Variable nur einmal benötigt werden.

```
WITH SCHEIN1.WAGEN DO
  BEGIN
    FOR I:= 1 TO 3 DO
      KREIS[I]:= ' ';
    KREIS[0]:=CHR(3);
  END;
```

ist also schneller als die Anweisungsfolge

```
FOR I:= 1 TO 3 DO
  SCHEIN1.WAGEN.KREIS[I]:=' ';
SCHEIN1.WAGEN.KREIS[0]:=CHR(3);
```

Komplexe Datenstrukturen mit Arrays und Records

In Bild 65 wurden Arrays und Records als Teile von Records deklariert. Natürlich ist es auch erlaubt, Arrays mit Records als Elementen zu benutzen:

```
VAR ZULASSUNGEN : ARRAY[1..200] OF KRAFTFAHRZEUGSCHEIN;
ZULASSUNGEN[30].WAGEN.KREIS:= 'B';
```

Dies ist einer der Gründe für die Flexibilität der Sprache Pascal. Die Standard- und Aufzählungstypen bilden die elementaren Bausteine, mit denen man je nach Bedarf hierarchisch strukturierte zusammengesetzte Datentypen definiert.

Eine dreidimensionale Netzgrafik

Als ein vollständiges Beispielprogramm für die Verwendung von Records zeigt Bild 67 die Darstellung einer dreidimensionalen Netzgrafik. Grundsätzlich ist das Zeichnen eines solchen Netzes sehr einfach: In zwei geschachtelten FOR-Anweisungen (ROW, POINT) berechnet man N * N Stützstellen. Jede Stützstelle wird mit ihrem (bereits gezeichneten) »linken« und »vorderen« Vorgänger durch eine Linie verbunden. Dabei müssen nur der erste Punkt jeder X- und Y-Reihe gesondert behandelt werden.

Aus den gegebenen X- und Y-Koordinaten wird nun mit einer (beliebigen) Funktion F eine zugehörige Z-Koordinate berechnet. Diese drei reellen Koordinaten zusammen werden mit einer Formel, die die perspektivische Verzerrung der Y-Achse berücksichtigt, in ganzzahlige Bildschirmkoordinaten umgewandelt.

Behandlung »verdeckter« Linien

Das Programm in Bild 67 ist etwas umfangreicher, da einerseits einige Zwischenergebnisse aus Geschwindigkeitsgründen aus den inneren FOR-Schleifen ausgelagert wurden und andererseits »verdeckte« Linien nicht gezeichnet werden. Normalerweise stellt das Hidden-line-Problem sehr hohe Anforderungen an die Rechenleistung des Computers. In diesem Fall ist jedoch die relative Lage der Punkte zueinander bereits bekannt. Um zu bestimmen, wie zwei Punkte P und Q verbunden werden, genügt in diesem Fall die Kenntnis der Sichtbarkeit dieser beiden Punkte (siehe Prozedur DRAWLINE).

Bedeutung des Typs VISIBILITY

Ein Punkt kann folgende Lagen besitzen (Aufzählungstyp VISIBILITY):

ABOVE	Er liegt oberhalb der durch das Netz gebildeten Oberfläche.
BELOW	Er liegt unterhalb der durch das Netz gebildeten Oberfläche.
HIDDEN	Er wird von der durch das Netz gebildeten Oberfläche verdeckt.
BORDER	Er liegt am vorderen oder rechten Rand des Netzes und ist daher in jedem Fall sichtbar.

Algorithmus

Insgesamt gibt es damit beim Verbinden der Punkte P und Q 16 verschiedene Kombinationen der Sichtbarkeit von P und Q. Diese werden mit der *Brute-force*-Methode in der Prozedur DRAWLINE geprüft. Dabei ist es nötig, für jeden Punkt die höchste und niedrigste Y-Koordinate zu kennen, die gerade noch durch die Oberfläche des Netzes verdeckt wird (YMAX, YMIN). Daher wird jeweils für die zuletzt gezeichnete Reihe und den neu berechneten Punkt ein Record mit folgenden Feldern benötigt:

```
TYPE LOCATION = RECORD
                  X,Y       : INTEGER;
                  WHERE     : VISIBILITY;
                  YMAX,YMIN: INTEGER
                END;
```

Mit diesen Erläuterungen können Sie das Programm in Bild 67 sicher nachvollziehen.

```
PROGRAM NETZ (INPUT,OUTPUT,GRAPHIC);
(* DREIDIMENSIONALE NETZGRAFIK MIT EINFACHEM 'HIDDEN-LINE'*)
(* ALGORITHMUS.                                            *)
  CONST MAXN = 100; (* MAX. ANZAHL AN PUNKTEN IN X-        *)
                    (* RICHTUNG - 1                        *)
        ORGX = 0;   (* LAGE LINKE VORDERE ECKE AUF BILD    *)
        ORGY = 150; (* LAGE LINKE VORDERE ECKE AUF BILD    *)
        MAXX = 320; (* MAXIMALE BILDSCHIRMKOORDINATE IN X *)
        MAXY = 400; (* MAXIMALE BILDSCHIRMKOORDINATE IN Y *)
        YXSCALE = 0.5;(* LAENGE DER PROJEKTION EINER       *)
                      (* STRECKE DER LAENGE 1 AUF DER Y-   *)
                      (* ACHSE IN X-RICHTUNG               *)
        YYSCALE = 0.5;(* DSGL. IN Y-RICHTUNG               *)
        BACK    = #157 #157 #157; (* CURSOR 3 * <--       *)

  TYPE  VISIBILITY = (ABOVE, HIDDEN, BELOW, BORDER);
        LOCATION   =
                   RECORD
                     X,Y  : INTEGER;     (* BILDSCHIRMKOORD.*)
                     WHERE: VISIBILITY;  (* SICHTBARKEIT    *)
                     YMAX : INTEGER;     (* BISHERIGE MAXIMA*)
                     YMIN : INTEGER;     (* UND MINIMA      *)
                   END;

  VAR   P: ARRAY[0..MAXN] OF       (* NUR EINE ZEILE! *)
             LOCATION;
        N: INTEGER;  (* ANZAHL DER PUNKTE IN X/Y RICHTUNG *)

        XU,XO,YU,ZM: REAL;      (* DARZUSTELLENDER BEREICH *)
        ROW, POINT  : INTEGER;  (* LFD. POSITION IM NETZ   *)
        ISFIRST     : BOOLEAN;  (* IN DER ERSTEN REIHE TRUE*)
```

Bild 67: Das Programm NETZ

```
      NEW          : LOCATION;                (* LFD. PUNKT *)
      D            : REAL;     (* SCHRITTWEITE              *)
      YAKT,XAKT    : REAL;     (* LFD. FUNKTIONSWERTE       *)
      SIZE         : REAL;     (* PIXEL-ABSTAND IN X        *)
      XADD, YADD   : INTEGER;  (* OFFSET IN PIXELN FUER     *)
                               (* DIE LFD. ZEILE            *)

FUNCTION F(X,Y:REAL): REAL;
(* DARZUSTELLENDE FUNKTION. ABBILDUNG R * R --> R *)
  CONST EPS = 1.0E-5;
  VAR   D : REAL;
BEGIN
  D:= SQRT(X*X+Y*Y);
  IF D<=EPS THEN F:=1.0
            ELSE F:= SIN(D) / D;
END; (* F *)

PROCEDURE DRAWLINE(VAR P,Q: LOCATION);
(* VERBINDE P MIT Q, FALLS LINIE SICHTBAR IST              *)
BEGIN
  CASE P.WHERE OF
    ABOVE : IF Q.WHERE=ABOVE THEN
              DRAW(1,P.X,P.Y,Q.X,Q.Y)
            ELSE
              DRAW(1,P.X,P.Y,Q.X,Q.YMAX);
    HIDDEN: CASE Q.WHERE OF
              BORDER,
              ABOVE : DRAW(1,P.X,P.YMAX,Q.X,Q.Y);
              HIDDEN:;(* NICHTS *)
              BELOW : DRAW(1,P.X,P.YMIN,Q.X,Q.Y);
            END;
    BELOW : IF Q.WHERE=BELOW THEN
              DRAW(1,P.X,P.Y,Q.X,Q.Y)
            ELSE
              DRAW(1,P.X,P.Y,Q.X,Q.YMIN);

      BORDER: DRAW(1,P.X,P.Y,Q.X,Q.Y);
    END; (* CASE *)
  END; (* DRAWLINE *)

BEGIN
  SCNCLR;
  WRITELN('ANZAHL DER STUETZSTELLEN IN');
  WRITE('X- UND Y-RICHTUNG:  30',BACK); READLN(N);
  WRITELN; N:= N-1;
  WRITE('X VON    : -10',BACK); READLN(XU);
  WRITE('X BIS    :   0',BACK); READLN(XO);
  WRITE('Y VON    : -10',BACK); READLN(YU);
  WRITE('Z FAKTOR: 150',BACK); READLN(ZM);

  GRAPHIC(1,1);

  D:= (XO-XU)/N; (* SCHRITTWEITE *)
  SIZE:= (MAXX-ORGX) / ((1.0 + YXSCALE)*N);

  ISFIRST:= TRUE; YAKT:= YU;
  (* ZEILENWEISE Y VON VORNE NACH HINTEN ERHOEHEN:          *)
  FOR ROW:= 0 TO N DO
    BEGIN
      (* BERECHNE KONSTANTEN FUER DIESE REIHE:              *)
      XADD:= ORGX + INT(ROW*SIZE*YXSCALE);
      YADD:= ORGY - INT(ROW*SIZE*YYSCALE);
      (* = OFFSET NACH RECHTS UND UNTEN *)
```

Bild 67: Das Programm NETZ (Fortsetzung nächste Seite)

```
XAKT:=XU;
FOR POINT:= 0 TO N DO
 WITH NEW DO
    BEGIN
      (* BERECHNE PUNKT IN BILDSCHIRMKOORDINATEN     *)
      X:= INT(POINT * SIZE) + XADD;
      Y:= YADD - INT( F(XAKT,YAKT) * ZM);

      IF (POINT=N) OR ISFIRST THEN
        BEGIN
          YMAX:= Y; YMIN:= Y; WHERE:= BORDER
        END
      ELSE
        BEGIN
        (* BESTIMME EXTREMWERTE AUS DEN ALTEN EX-    *)
        (* TREMWERTEN UND DER AKTUELLEN Y-KOORDINATE*)
        (* DARAUS FOLGT AUCH DIE SICHTBARKEIT        *)
        YMAX:= (P[POINT].YMAX+P[POINT+1].YMAX) DIV 2;
        YMIN:= (P[POINT].YMIN+P[POINT+1].YMIN) DIV 2;
        IF Y>YMAX THEN
          BEGIN
            YMAX:= Y; WHERE:=ABOVE;
          END
        ELSE
          IF Y<YMIN THEN
            BEGIN
              YMIN:= Y; WHERE:=BELOW;
            END
          ELSE
            WHERE:= HIDDEN;
        END; (* WITH *)

      (* VERBINDE PUNKT MIT NACHBARN, SOWEIT VOR- *)
      (* HANDEN UND SICHTBAR                      *)
      IF NOT ISFIRST THEN          (* NACH 'VORNE' *)
        DRAWLINE(P[POINT],NEW);
      IF POINT<>0 THEN             (* NACH 'LINKS' *)
        DRAWLINE(P[POINT-1],NEW);
            P[POINT]:= NEW;
          XAKT:= XAKT + D;                (* NAECHSTER PUNKT *)
        END; (* FOR POINT *)
    YAKT:= YAKT+D;                        (* NAECHSTE REIHE  *)
    ISFIRST:= FALSE;
  END; (* FOR ROW *)

  REPEAT UNTIL KEYPRESSED;
  GRAPHIC(0);
END.
```

Bild 67: Das Programm NETZ (Schluß)

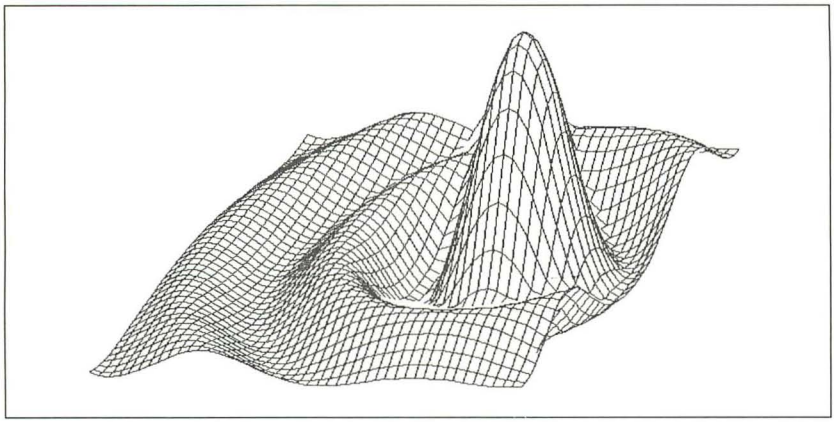


Bild 68: Ein dreidimensionales Netz

Aufgaben

Brüche als Records

1. Schreiben Sie ein Paket mit Prozeduren, das die Bearbeitung von Bruchzahlen ermöglicht. Dabei sollen Brüche nicht als Zahlen vom Typ REAL dargestellt werden, sondern als Paare ganzer Zahlen:

```
TYPE BRUCH = RECORD
               ZAEHLER: INTEGER;
               NENNER : INTEGER;
             END;
```

Damit der Zahlenbereich der ganzen Zahlen nicht bei den einfachsten Operationen überschritten wird, sollen Zähler und Nenner immer gekürzt vorliegen (also $^1/_{134}$ und nicht $^{2345}/_{314230}$). Zum Kürzen können Sie die Funktion GGT aus Abschnitt 2.11 verwenden.

```
PROCEDURE KUERZE(VAR ZAEHLER,NENNER: INTEGER);
PROCEDURE LESEN(VAR B: BRUCH);
PROCEDURE SCHREIBEN(B: BRUCH);
PROCEDURE PLUS(A,B: BRUCH; VAR C:BRUCH);
PROCEDURE MAL (A,B: BRUCH; VAR C: BRUCH);
PROCEDURE KEHRWERT(A: BRUCH; VAR C: BRUCH);
FUNCTION GROESSER(A,B:BRUCH): BOOLEAN;
FUNCTION GLEICH(A,B: BRUCH): BOOLEAN;
FUNCTION WERT(A:BRUCH): REAL;
(* Liefert den reellen Quotienten ZAEHLER/NENNER *)
```

Zwei Beispiele sollen Ihnen das Prinzip verdeutlichen:

```
PROCEDURE MAL (A,B:BRUCH; VAR C: BRUCH);
(* C:= A * B. Da A und B jeweils gekürzt vorliegen, *)
(* genügt ein kreuzweises Kürzen vor der Mult.      *)
BEGIN
```

```
  KUERZE(A.ZAEHLER, B.NENNER);
  KUERZE(B.ZAEHLER, A.NENNER);
  C.ZAEHLER:= A.ZAEHLER * B.ZAEHLER;
  C.NENNER := A.NENNER  * B.NENNER;
  (* Zähler und Nenner von C sind jetzt teilerfremd *)
END; (* MAL *)

PROCEDURE SCHREIBEN(A: BRUCH);
(* Ausgabe des Bruches A *)
BEGIN
  WITH A DO
    IF NENNER = 1 THEN WRITE(ZAEHLER)
    ELSE
      WRITE(ZAEHLER, '/', NENNER);
END; (* SCHREIBEN *)
```

2. Sollten Sie ab und zu mit komplexen Zahlen arbeiten (müssen), ist es vielleicht interessanter, den Typ KOMPLEX mit seinen Operationen zu implementieren

   `TYPE KOMPLEX = RECORD RE,IM: REAL; END;`

   Als Operationen bieten sich Addition, Multiplikation, Bildung der konjugiert komplexen Zahl, Betragsfunktion sowie die Umwandlung in Polarkoordinaten an.

Sortieren mit Records

3. Bisher bestand bei allen Sortierprogrammen das zu sortierende Array alleine aus den Schlüsseln, die sortiert werden sollten. Normalerweise möchte man jedoch Daten sortieren, bei denen die Elemente die Struktur eines Records besitzen.

```
TYPE ELEMENT = RECORD
                 NAME     : STRING[30];
                 EINKOMMEN: REAL;
                 STADT    : STRING[30];
               END;
VAR A: ARRAY OF ELEMENT;
```

   Ändern Sie die Sortierprogramme BUBBLESORT bzw. QUICKSORT, so daß die Daten nach folgenden Kriterien sortiert werden:

   3.1 Namen alphabetisch aufsteigend
   3.2 Einkommen absteigend
   3.3 Namen alphabetisch aufsteigend. Bei Namensgleichheit weiter nach Städten aufsteigend.

Bei den Austauschoperationen werden weiterhin ganze Elemente zugewiesen, während die Vergleichsoperationen auf die Schlüsselfelder (NAME, EINKOMMEN bzw. STADT) eingeschränkt werden müssen.

2.15 Variante Records

Beispiel für variante Datenstrukturen

Eine in der Praxis recht häufige Eigenschaft von Datensätzen ist es, für einzelne Ausprägungen der Daten **unterschiedliche** Merkmale zu enthalten. Als Beispiel betrachte man grafische Objekte. Die Aufgabe besteht darin, ein Bild durch Zusammenstellung von grafischen Primitiven (Rechtecke, Kreise, Linien und Texte) zu beschreiben (s. Deklaration in Bild 69).

```
CONST MAXOBJ = 50;
TYPE  TPRIMITIV = (RECK, BLOCK, KREIS, LINIE,
                   TEXT, HINTERGRUND);
      TKOORD    = RECORD X,Y:INTEGER END;
      TOBJEKT   = RECORD
                    FARBE : 1..3; (* ROT, GRUEN, BLAU *)
                    INTENS: 0..1; (* HELL, DUNKEL     *)
                    CASE ART: TPRIMITIV OF
                     RECK, BLOCK:
                      (RECHTSUNTEN,LINKSOBEN: TKOORD);
                     KREIS:
                      (MITTE : TKOORD;
                       RADIUS: INTEGER);
                     LINIE:
                       (VON, BIS: TKOORD);
                     TEXT:
                       (POS1 : TKOORD;
                        STRNG: STRING[10]);
                     HINTERGRUND: ();
                  END;
VAR  BILD: ARRAY[1..MAXOBJ] OF TOBJEKT;
     OBJEKT: TOBJEKT;
```

Bild 69: Record-Deklaration mit Varianten

Syntax

In der Deklaration werden Rechtecke (RECK), ausgemalte Rechtecke (BLOCK), Kreise, Linien und Texte berücksichtigt. Alle Objekte besitzen gemeinsame Merkmale: die Farbe (FARBE), die Helligkeit (INTENS) und eine Kennzeichnung (ART), die angibt, um welches elementare Objekt es sich handelt. Diese gemeinsamen Felder werden wie in einem einfachen Record definiert.

Fester und varianter Teil eines Records

An diesen **festen** Teil schließt sich der **variante** Teil an: Er wird durch das Wortsymbol CASE eingeleitet. Diesem folgt das sogenannte Auswahlfeld (engl. *tagfield*). In diesem speziellen Beispiel ist dies das Feld ART. Der Typ des Auswahlfeldes muß ein skalarer Typ (also nicht REAL oder STRING) sein, der durch einen Typ-**Namen** angegeben wird.

Nach dem Wortsymbol OF werden die einzelnen Varianten aufgeführt, die in Abhängigkeit von dem aktuellen Wert des Auswahlfeldes (ART) gültig sind. Die jeweiligen Konstanten des Typs (TPRIMITIV) werden durch Kommata getrennt. Nach einem Doppelpunkt folgen in Klammern die Felder für diese Variante. Die Struktur dieser Feldliste zwischen den Klammern entspricht genau der Feldliste zwischen den Wortsymbolen RECORD und END. Also könnten dort geschachtelt wiederum Varianten stehen.

Alle Varianten sind durch Semikola getrennt. Bitte beachten Sie, daß der variante Teil nach dem Symbol CASE **nicht** durch ein eigenes END (wie bei der Anweisung CASE im Anweisungsteil) beendet wird. Deshalb steht in Bild 69 am Ende von TOBJEKT nur ein einzelnes END, das die mit RECORD begonnene Struktur beendet.

Gültigkeit der Varianten

Wie durch die Struktur der Deklaration bereits suggeriert wird, ist zu jedem Zeitpunkt nur **eine einzige** Variante (RECK oder BLOCK oder KREIS oder ...) gültig. Nach der Zuweisung

```
OBJEKT.ART:= KREIS
```

wären also neben den Feldern FARBE, INTENS nur die Felder der Variante KREIS gültig, die man dann wie folgt belegen kann:

```
OBJEKT.MITTE.X := 30;
OBJEKT.MITTE.Y := 30;
OBJEKT.RADIUS  := 15:
```

Die Felder der übrigen Varianten (wie z.B. POS1 und VON) dürften bei dem Wert KREIS des Auswahlfeldes ART nicht verwendet werden. Erwähnenswert ist noch die Tatsache, daß die Feldliste zu einer Varianten leer sein kann:

```
HINTERGRUND:()
```

Ist also OBJEKT.ART=HINTERGRUND, so sind nur die festen Felder (FARBE, INTENS) relevant. Die Nennung leerer Varianten dient nur der Dokumentation der Struktur und ist syntaktisch nicht verpflichtend. Wie üblich, finden Sie die exakte Syntax (varianter) Records in Anhang A, und zwar im Syntaxdiagramm FELDLISTE.

Es ist ein schwerer Programmierfehler, auf Varianten zuzugreifen, die nicht dem aktuellen Wert des Auswahlfeldes entsprechen:

Falsch!

```
WITH OBJECT DO
  BEGIN ART:=TEXT; VON.X:= 30; END;
```

CASE-Anweisung für Operationen mit varianten Records

Um solche Probleme zu vermeiden, bietet sich bei Operationen mit varianten Records die Verwendung einer Fallunterscheidung (CASE-Anweisung) an, die die Struktur des varianten Records reflektiert:

```
PROCEDURE READOBJECT (OBJEKTART: TPRIMITIV; VAR O: TOBJECT);
(* BENUTZEREINGABE EINES OBJEKTES VOM TYP TOBJECT *)

  PROCEDURE READK(WAS: STRING; VAR K: TKOORD);
  (* EINGABE EINES KOORDINATENPAARES *)
  BEGIN
    WRITE(WAS, ' (X Y) =');
    WITH K DO
      READLN(X,Y);
  END; (* READK *)

BEGIN
  WITH OBJECT DO
    BEGIN
      WRITE('Farbe :')      ; READLN(FARBE);
      WRITE('Intensität :'); READLN(INTENS);
      ART:= OBJEKTART;
      CASE OBJEKTART OF
        RECK,BLOCK:
          BEGIN
            READK('rechte untere Ecke', RECHTSUNTEN);
            READK('linke obere Ecke', LINKSOBEN);
          END;
        KREIS:
          BEGIN
            READK('Mitte', MITTE);
            WRITE('Radius :'); READLN(RADIUS);
          END;
        LINIE:
          BEGIN
            READK('Anfangspunkt', VON);
            READK('Endpunkt', BIS);
          END;
        TEXT:
          BEGIN
```

Bild 70: Eingabe eines varianten Records (Fortsetzung nächste Seite) ►

```
            READK('Position 1. Zeichen', POS1);
            WRITE('Text (<=10 Zeichen):'); READLN(STRING);
          END;
        HINTERGRUND: BEGIN (* NICHTS *) END
      END; (* CASE *)
    END; (* WITH *)
  END; (* READOBJECT *)
```

Bild 70: Eingabe eines varianten Records (Schluß)

Eingabenvariante Records

Für jeden Record-Typ (TKOORD und TOBJEKT) existiert also eine separate Prozedur zur Eingabe. Natürlich werden die eingelesenen Daten von den Prozeduren über *Variablenparameter* als Ergebnisse zurückgeliefert. Innerhalb der CASE-Anweisung wird die durch den *Wertparameter* OBJEKTART definierte Variante eingelesen. In dieser Fallunterscheidung ist (im Gegensatz zur Deklaration des Records) die Angabe der *Leeranweisung* hinter der Fallmarke HINTERGRUND obligatorisch. Ansonsten würde zur Laufzeit bei dem Wert OBJEKTART=HINTERGRUND das Programm mit einer Fehlermeldung abgebrochen:

```
NO LABEL FOR CASE ERROR
```

Speicherverteilung für Records

Gerade bei großen Datenstrukturen im Hauptspeicher ist es auch für den Programmierer in einer problemorientierten »Hochsprache« sinnvoll, die Repräsentation der Daten im Computer zu kennen. Deshalb soll die Speicherverteilung (in *Pascal 2.0*) für variante Records kurz umrissen werden. Die exakte Definition der Speicherverteilung für beliebige Typen finden Sie in der Dokumentation in Abschnitt 4.4.2. Da zu einem Zeitpunkt das *tagfield* (Auswahlfeld) nur einen Wert annehmen kann, erhalten alle Varianten denselben Speicherplatz. Die Größe eines Objektes vom Typ TOBJEKT wird durch die Größe des festen Teils (FARBE und INTENS) zuzüglich der Größe der längsten Variante bestimmt. Damit ergibt sich die in Bild 71 skizzierte Speicherverteilung.

FARBE			
INTENS			
ART			
RECHTSUNTEN.X	MITTE.X	VON.X	POS1.X
RECHTSUNTEN.Y		VON.Y	
LINKSOBEN.X	RADIUS	BIS.X	STRNG[0..1]
LINKSOBEN.Y		BIS.Y	STRNG[2..3]
			STRNG[4..5]
FREI	FREI	FREI	STRNG[6..7]
			STRNG[8..9]
			STRNG[10]

Bild 71: Struktur des Typs TOBJEKT

Optimierung des Speicherplatzbedarfs durch Auslagerung langer Varianten

In diesem Fall ist TEXT die längste Variante. Alle anderen Varianten werden ebenfalls mit dieser Größe gespeichert und belegen daher ungenutzten Speicherplatz. Will man sehr große Arrays mit solchen Objekten bilden, so kann es sinnvoll sein, die größte Variante zu kürzen. Eine Möglichkeit besteht darin, das Feld STRNG auszulagern und durch einen Verweis in eine weitere Tabelle mit Strings zu ersetzen:

```
TYPE STRINGREF= 1..MAX;
VAR  STRINGARRAY : ARRAY [STRINGREF] OF STRING[10];
```

Man müßte außerdem die Variante TEXT im Record TOBJEKT folgendermaßen ändern:

```
TEXT:
  (POS1: TKOORD;
   STRNG: STRINGREF);
```

Um einen Text im Array BILD (siehe Deklaration in Bild 69) an der I-ten Stelle einzufügen, speichert man zunächst den String an einer freien Position im STRINGARRAY (z.B. J-tes Element). Dem Feld STRNG im Record wird dann nur der Index J zugewiesen:

```
WITH BILD[I] DO
  BEGIN
    FARBE := 3;
    INTENS:= 2;
    ART   := TEXT;
    POS1.X:= 0; POS1.Y:= 3;
    (* String eintragen *)
    STRINGARRAY[J]:= 'Beispieltext';
    (* Referenz notieren *)
    STRNG:=J;
  END;
```

Damit ergibt sich eine kompaktere Speicherung der varianten Records, wie sie in Bild 72, Seite 162, dargestellt ist.

Solche Speicherplatzoptimierungen sind gerade auf Mikrocomputern erforderlich und holen den angehenden *software engineer* allzu rasch von abstrakten objektorientierten Datenmodellen auf den Boden der Realität aus Bits und Bytes zurück.

Variante Records ohne tagfield

Der Vollständigkeit halber sei noch erwähnt, daß es in Pascal zulässig ist, anstelle des Auswahlfeldes nur einen Typbezeichner zu nennen. Die aktuell gültige Variante wird also **nicht** in einem *tagfield* gespeichert. Statt dessen muß der Programmierer aus dem Kontext herleiten, welche Variante des Records momentan gültig ist. Diese Records werden praktisch nur zu *schmutzigen* Operationen verwendet, die normalerweise (aus gutem Grund) in Pascal verboten sind: Bei diesen Operationen nutzt man (wie bei den *common areas* in Fortran)

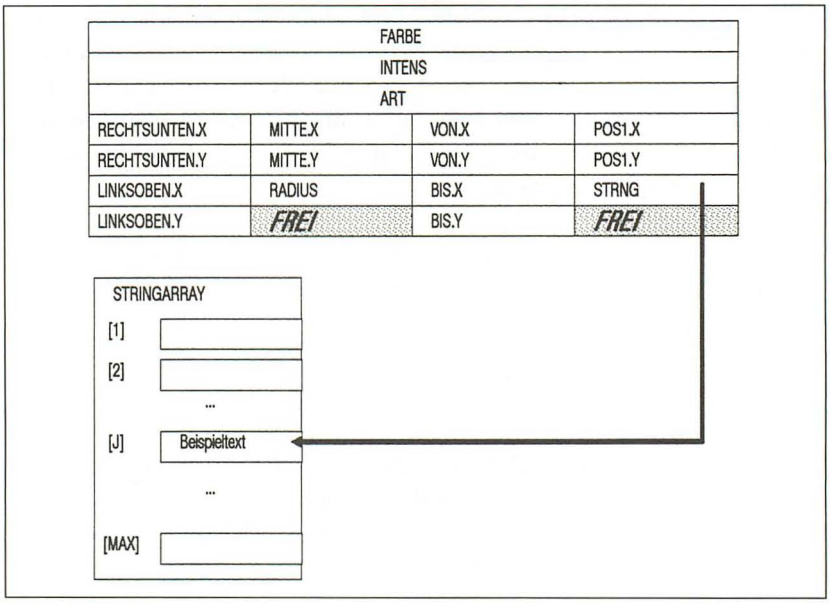


Bild 72: Gepackter varianter Record

die Tatsache aus, daß die einzelnen Varianten denselben Speicherplatz belegen, so daß die gleiche binäre Codierung bei verschiedenen Typen eine unterschiedliche Bedeutung besitzt. Daher kann man mit der folgenden Anweisung einer Variablen eines Aufzählungstyps einen Wert zuweisen, dessen Ordinalwert gleich 3 ist:

```
PROGRAM NASTY (OUTPUT);

  TYPE AUFZAEHL = (ROT, GRUEN, BLAU, SCHWARZ);
  VAR  SCHLIMM = RECORD
                   CASE BOOLEAN OF
                     TRUE:  (I: INTEGER);
                     FALSE: (V: AUFZAEHL);
                 END;
BEGIN
  SCHLIMM.I:=3;
  IF SCHLIMM.V = SCHWARZ THEN
    WRITELN('SCHLIMM.V IST SCHWARZ');
END.
```

Da solche Operationen inhärent von Eigenschaften spezieller Computer und Compiler abhängig sind, sollten Sie diese nicht in Ihren Programmen verwenden. In *Pascal 2.0* gibt es explizite Operationen für solche Typumwandlungen, die in der Dokumentation behandelt werden.

Bottom-up-Definition zusammengesetzter Typen

Bild 69 zeigt exemplarisch, wie man in Pascal Daten-Deklarationen strukturiert. Angefangen bei den skalaren Bausteinen (Indexgrenzen, Aufzählungstypen) bildet man eine höhere Abstraktionsstufe: Man arbeitet zum Beispiel mit Koordinaten und nicht mehr mit INTEGER-Zahlen. Anschließend kann man diese abstrakteren Typen noch zu Records und Arrays (Verbunden und Feldern) geeigneter Dimensionen zusammenfassen.
Diese Vorgehensweise (von unten nach oben, *bottom up*) ist zwingend notwendig. Da der Compiler den Text in einem Durchgang (*one pass*) liest, muß jeder Typname vor der ersten Anwendung bereits deklariert sein. Konkret bedeutet dies, daß die Deklaration des Typs TKOORD vor der Anwendung des Typs in der Deklaration von TOBJEKT erfolgen muß.

2.16 Der Datentyp File

Sequentielle Dateien in Pascal

Alle bisher behandelten Daten(typen) besitzen eine fest definierte konstante Größe. Ein Vorteil von Variablen dieser Typen ist, daß sie jederzeit im Programm direkt über ihren Namen (evtl. gefolgt von einem Index in eckigen Klammern oder einem Feldnamen nach einem Punkt) angesprochen werden können. In der Definition der Sprache Pascal wird jedoch auch eine Datenstruktur angegeben, deren Größe während der Laufzeit eines Programmes variabel ist. Dafür ist der Zugriff auf Elemente der Struktur nur in fester Reihenfolge mit speziellen Prozeduren möglich. Diese Struktur heißt **File** und formalisiert das Konzept der sequentiellen Dateien.

Deklaration

Zu jedem Typ T läßt sich mit FILE OF T ein File mit dem Komponententyp T bilden. Beispiele für Deklarationen von File-Variablen sind in Bild 73 gegeben. Dabei werden Typbezeichner benutzt, die bereits in Bild 65 und 69 definiert wurden. ADRESSBUCH ist also eine File-Variable mit Komponenten vom Typ ADRESSE.

```
TYPE (* siehe Bild 65 und Bild 69 *)
VAR  ZAHLENSPEICHER = FILE OF INTEGER;
     KOEFFIZIENTEN  = FILE OF REAL;
     TEXTFILE       = FILE OF CHAR;
     BILDDATEI      = FILE OF TOBJEKT;
     ADRESSBUCH     = FILE OF ADRESSE;
```

Bild 73: Deklaration von File-Variablen

Das englische Wort *file* bezeichnet ursprünglich einen Ordner oder eine Sammelmappe. In der Datenverarbeitung wird *file* am besten mit **Datei** übersetzt. Dateien sind Folgen (gleichartiger) Daten, die meist auf externen Massenspeichern abgelegt werden. Da die Verwaltung von Dateien auf verschiedenen Computern sehr unterschiedlich realisiert wird, beschränkt sich Pascal auf elementare Operationen mit sequentiellen Dateien. Dennoch

weichen viele Implementierungen der Sprache vom nachfolgend (und im *report*) beschriebenen Standard ab.

Struktur eines Files und die Puffervariable

`VAR F: FILE OF T;`

deklariert eine File-Variable F. Diese Variable besteht aus einer (eventuell leeren) Folge von Komponenten des Typs T. Somit besteht die Variable BILDDATEI aus einer Folge von Records des Typs TOBJEKT. Die Anzahl der Komponenten innerhalb eines Files ist veränderlich. Der Zugriff auf die Komponenten kann nur sequentiell lesend oder sequentiell schreibend erfolgen. Zu jedem Zeitpunkt während der Programmausführung ist nur eine Komponente von F sichtbar. Sie wird mit F↑ wie eine »normale« Variable in Zuweisungen und Ausdrücken bezeichnet. F↑ wird auch die *Puffervariable* des Files F genannt.

2.16.1 Sequentiell schreiben

Mit dem Prozeduraufruf REWRITE(F) wird F zum Schreiben vorbereitet. Alle Komponenten der Variablen F werden gelöscht. Nun kann man der Puffervariablen F↑ einen Wert vom Typ T zuweisen. Durch den Prozeduraufruf PUT(F) wird der Inhalt der Puffervariablen als eine neue Komponente in F aufgenommen. Jeder Aufruf PUT(F) erweitert F um eine Komponente. Die Komponenten werden in der Reihenfolge gespeichert, in der sie mit PUT erzeugt wurden.

Erzeugen eines Files

Diese abstrakten Regeln sollen ein kleines Programmstück und Bild 75 erläutern. Es werden nacheinander zwei Zahlen (99 und 43) auf das File D gespeichert. Da die Datei D ganze Zahlen speichern soll, wird sie mit dem Typ FILE OF INTEGER deklariert. Nachdem das File zum Schreiben eröffnet wurde, ist das File leer und der Inhalt der Puffervariablen D↑ undefiniert (Bild 75.0). Nun werden die zu schreibenden Zahlen zunächst der Puffervariablen zugewiesen und dann mit PUT(D) der Inhalt von D↑ auf das File geschrieben. Damit ergibt sich der in Bild 75.1-3 dargestellte Inhalt von D. Dabei wird die Puffervariable als »Fenster« in das File dargestellt.

```
PROGRAM WRITEFILE (INPUT,OUTPUT);
(* Dieses Programm muß in PASCAL 2.0 umformuliert werden! *)
(* s. Text!                                                *)
  VAR D: FILE OF INTEGER;
BEGIN
  REWRITE(D);          (* --- 0 --- *)
  D^ := 99;            (* --- 1 --- *)
  PUT(D); D^ := 43;    (* --- 2 --- *)
  PUT(D);              (* --- 3 --- *)
END.
```

Bild 74: File schreiben

2.16.2 Sequentiell lesen

Lesen eines existierenden Files

Um die in einer File-Variablen gespeicherten Daten wieder zu lesen, muß man zunächst F mit dem Prozeduraufruf RESET(F) zum lesenden Zugriff vorbereiten. Dabei wird der Puffervariablen F↑ die erste Komponente in F als Wert zugewiesen. Dieser Wert kann nun beliebig in Ausdrücken verwendet werden. Durch den Aufruf von GET(F) wird der Wert der nächsten Komponente von F nach F↑ übertragen. Somit kann man durch eine Folge von Aufrufen der Prozedur GET jede Komponente in F erreichen. Die Funktion EOF(F) (engl. *end of file*, Dateiende) liefert den Wert TRUE, falls beim sequentiellen Lesen das Ende des Files F erreicht wurde. Dann ist der Inhalt der Puffervariablen F↑ **undefiniert**. Beim Schreiben mit REWRITE und PUT ist EOF(F) immer TRUE.

Die Funktion EOF

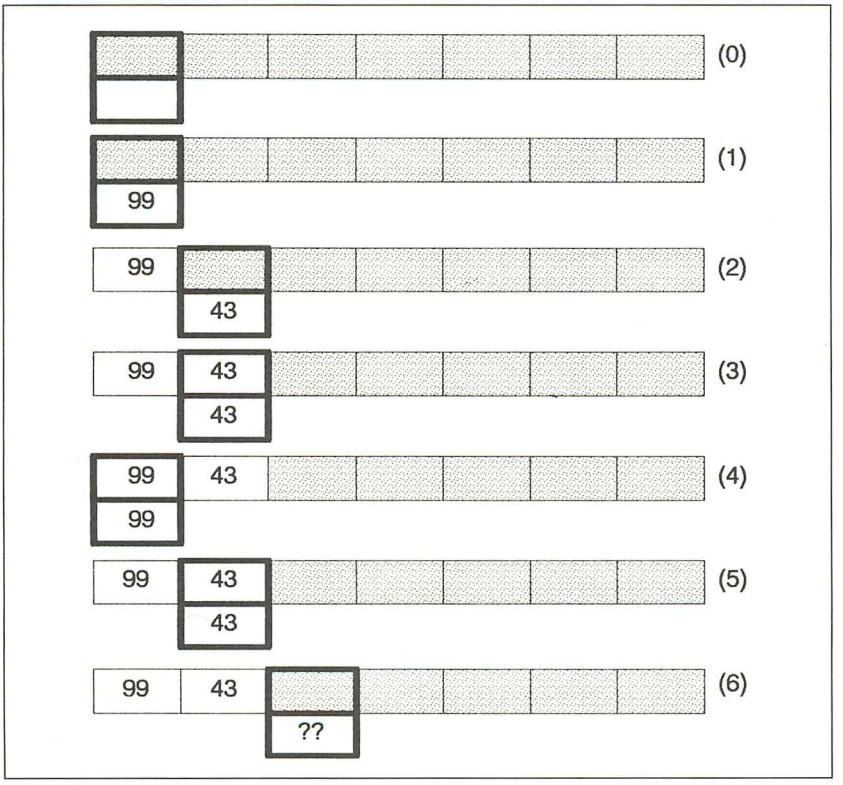


Bild 75: Darstellung der Dateioperationen

Ein kleines Programmstück und Bild 75 zeigen das Lesen der im Abschnitt 2.16.1 erstellten sequentiellen Datei D.

```
PROGRAM READFILE (INPUT,OUTPUT);
(* Dieses Programm muß in PASCAL 2.0 umformuliert werden! *)
(* s. Text!                                                *)
  VAR D: FILE OF INTEGER;
BEGIN
  RESET(D);          (* --- 4 --- *)
  WRITE(D↑);
  GET(D);            (* --- 5 --- *)
  WRITE(D↑);
  GET(D);            (* --- 6 --- *)
  IF EOF(D) THEN
    WRITELN('Dateiende erreicht!');
END.
```

Bild 76: File lesen

Regeln für File-Operationen

Ein Wechsel zwischen Lesen und Schreiben auf ein File F ist in beliebiger Reihenfolge möglich. Dabei ist aber zu beachten, daß Lesezugriffe nur nach RESET und schreibende Zugriffe nur nach REWRITE möglich sind. Außerdem löscht jeder Aufruf von REWRITE alle eventuell zuvor in F enthaltenen Komponenten!

File-Typen können wie alle anderen Typen als Teil zusammengesetzter Datentypen (Record, Array) auftreten. Jedoch sind keine Files zulässig, deren Komponenten ebenfalls Files sind. Zuweisungen zwischen Variablen vom Typ File sind nicht erlaubt. Files dürfen nur als *Variablenparameter* an Prozeduren übergeben werden.

RESET und REWRITE existieren in Pascal 2.0 nicht

Der Datentyp File wird in *Pascal 2.0* praktisch wie oben beschrieben realisiert. Einen Unterschied bilden jedoch die Anweisungen RESET und REWRITE. Das Betriebssystem des C 128 erwartet nämlich genauere Angaben über jedes File. Man muß festlegen, auf welchem Peripheriegerät (Floppy, Datasette, Festplatte an externer Schnittstelle) die Speicherung der Komponenten erfolgt, welchen Namen das File auf dem Medium erhält etc. Daher werden die Prozeduren RESET und REWRITE durch die Prozeduren OPEN und CLOSE ersetzt.

Umwandlung von Programmen mit File-Operationen

An dieser Stelle wird eine Methode vorgestellt, wie Sie mit minimalem Aufwand Pascal-Programme aus der Literatur, die RESET und REWRITE benutzen, in *Pascal 2.0* umwandeln. Um die volle Mächtigkeit der Prozeduren OPEN und CLOSE zu nutzen, finden Sie aber in Kapitel 3 und der Dokumentation in Kapitel 4 genauere Hinweise zu diesen Standardprozeduren.

In *Pascal 2.0* wird eine Datei mit dem Prozeduraufruf

`OPEN(Filevariable, 8, 3, 'DATEN,SEQ,WRITE')`

zum Schreiben eröffnet. Damit werden alle Daten der File-Variablen in der

sequentiellen Datei mit dem Namen »DATEN« auf der Diskette gespeichert. Die Zahl 3 bildet eine sogenannte *Sekundäradresse*. Haben Sie mehrere File-Variablen in Ihrem Programm, so muß jede eine eigene Sekundäradresse (3, 4, 5, 6 bis 14) und einen eigenen File-Namen (»DATEN1«, »DATEN2« etc.) erhalten. Anschließend können Sie die Daten wie oben beschrieben mit PUT(*File-Variable*) ausgeben. Am Ende der Ausgabe muß außerdem die Datei geschlossen werden:

```
CLOSE(Filevariable)
```

Um eine Datei zum Lesen zu eröffnen, verwenden Sie ebenfalls die Prozedur OPEN. Wählen Sie dabei denselben File-Namen wie beim Schreiben. Die Sekundäradresse muß wiederum eindeutig aus dem Intervall 3 bis 14 gewählt werden:

```
OPEN(Filevariable, 8, 3, 'DATEN,SEQ,READ')
```

Nach diesem OPEN-Befehl ist der Inhalt der Puffervariablen noch undefiniert. Mit GET(*File-Variable*) können Sie die Datei wie oben beschrieben komponentenweise lesen. Auch nach dem Lesen muß eine Datei wieder geschlossen werden:

```
CLOSE(Filevariable)
```

Löschen eines existierenden Files

Bitte beachten Sie, daß bei aufeinanderfolgenden OPEN-Befehlen, die eine Datei F zum Schreiben eröffnen, die zuvor in F enthaltenen Komponenten nicht automatisch gelöscht werden. Sie müssen daher zuvor einen expliziten Löschbefehl angeben. Dies geschieht ebenfalls mit einer File-Variablen:

```
VAR COMMAND : FILE OF CHAR;
OPEN(COMMAND,8,15,'SO:DATEN'); CLOSE(COMMAND);
```

Damit sehen die Beispielprogramme aus Bild 74 und 76 in *Pascal 2.0* folgendermaßen aus:

```
PROGRAM WRITEREADFILE (INPUT,OUTPUT);
(* So sehen Bild 74 und 76 in PASCAL 2.0 aus *)
  VAR D       : FILE OF INTEGER;
      COMMAND: FILE OF CHAR;
BEGIN
  (* Zuerst evtl. vorhandene Daten löschen: *)
  OPEN(COMMAND,8,15,'SO:DATA'); CLOSE(COMMAND);

  OPEN(D,8,3,'DATA,SEQ,WRITE');      (* --- 0 --- *)
  D↑:= 99;                           (* --- 1 --- *)
  PUT(D); D↑:= 43;                   (* --- 2 --- *)
  PUT(D);                            (* --- 3 --- *)
  CLOSE(D);
```

Bild 77: File-Operationen in Pascal 2.0 (Fortsetzung nächste Seite) ►

```
  OPEN(D,8,3,'DATA,SEQ,READ');
  GET(D)                          (* --- 4 --- *)
  WRITE(D↑);
  GET(D);                         (* --- 5 --- *)
  WRITE(D↑);
  GET(D);                         (* --- 6 --- *)
  IF EOF(D) THEN
    WRITELN('Dateiende erreicht!');
  CLOSE(D);
END.
```

Bild 77: File-Operationen in Pascal 2.0 (Schluß)

Bearbeitung sequentieller Files

Da die Länge eines Files variabel ist, verwendet man bei der Ein- und Ausgabe sinnvollerweise WHILE- und REPEAT-Anweisungen. Das generelle Schema für die Ein- und Ausgabe sequentieller Dateien zeigt Bild 78. Dort werden die Quadratwurzeln der ersten 100 ganzen Zahlen in der File-Variablen SQUAREROOT gespeichert und nachfolgend gelesen.

```
PROGRAM ROOTS (INPUT,OUTPUT);
  VAR SQUAREROOT: FILE OF REAL;
      COMMAND   : FILE OF CHAR;
      I         : INTEGER;
BEGIN
  OPEN(COMMAND,8,15,'S0:SQRT'); CLOSE(COMMAND);
  OPEN(SQUAREROOT,8,3,'SQRT,SEQ,WRITE');
  FOR I:= 1 TO 100 DO
    BEGIN
      SQUAREROOT↑:= SQRT(I); PUT(SQUAREROOT);
    END;
  CLOSE(SQUAREROOT);

  OPEN(SQUAREROOT,8,3,'SQRT,SEQ,READ');
  GET(SQUAREROOT);
  WHILE NOT EOF(SQUAREROOT) DO
    BEGIN
      WRITELN(SQUAREROOT↑ :15:7); GET(SQUAREROOT);
    END;
  CLOSE(SQUAREROOT);
END.
```

Bild 78: Ein File mit reellen Komponenten

Sortieren sequentieller Files

Wie für Arrays existieren auch für Files zahlreiche verschiedene Sortierverfahren, die jeweils für spezielle Anwendungsgebiete optimale Ergebnisse liefern. Bei der Sortierung eines Files muß jedoch erschwerend die Tatsache berücksichtigt werden, daß die Daten nur in einer festen Reihenfolge nacheinander zur Verfügung stehen. Natürlich kann man ein File sequentiell in ein Array einlesen, es dort mit einem der Algorithmen für Arrays (z.B. QUICKSORT) sortieren und anschließend sortiert als File speichern. Jedoch existieren Anwendungsfälle, in denen die Datengröße die Hauptspeicherkapazität bei weitem übersteigt. Dann muß man auf die sogenannten *externen Sortierverfahren* ausweichen, die z.B. in [2] (siehe Anhang E) beschrieben werden.

Der Algorithmus für natürliches Mischsortieren

Eine der einfachsten Methoden zur Sortierung eines Files heißt *natürliches Mischsortieren*. Die Sortierung eines Files C erfolgt dabei in mehreren Durchläufen. Jeder Durchlauf gliedert sich in zwei Schritte:

1. Das File C wird Komponente für Komponente auf zwei weitere Files A und B verteilt (*distribute*).
2. Die beiden Files A und B werden gemischt (*merge*). Die Operation »Mischen« bezeichnet in der Datenverarbeitung das Zusammenfassen aufsteigender Teilsequenzen zu einer einzigen (längeren) aufsteigenden Teilsequenz. Das Ergebnis dieser Zusammenfassung wird wiederum in C gespeichert.

Diese beiden Schritte werden so lange wiederholt, bis C nur aus einer einzigen aufsteigend sortierten Sequenz besteht. In Bild 79 sind die nur in *Pascal 2.0* erforderlichen Erweiterungen mit Kommentaren gekennzeichnet.

```
PROGRAM MERGE (INPUT,OUTPUT);
(* NATUERLICHES MISCHSORTIEREN. (KAPITEL 2.16)              *)
(* IM PROGRAMM SIND DIE OPERATIONEN RESET UND REWRITE       *)
(* DURCH OPEN- UND CLOSE-SEQUENZEN ERSETZT WORDEN.          *)
(* 1.5.1986 FLORIAN MATTHES                                 *)
(* QUELLE: WIRTH; ALGORITHMEN & DATENSTRUKTUREN KAP.2.3.2 *)

 CONST NAMEA ='0:FILE.A';(* FILENAMEN FUER DIE FLOPPY      *)
       NAMEB ='0:FILE.B';
       NAMEC ='0:FILE.C';
       RD    =',SEQ,READ';
       WR    =',SEQ,WRITE';

 TYPE ITEM = RECORD
               KEY: INTEGER
               (* HIER KOENNEN WEITERE FELDER STEHEN *)
             END;

      TAPE = FILE OF ITEM;

  VAR  C        : TAPE;

   PROCEDURE ERASE(N: STRING);
   (* LOESCHE FILE N AUF DER DISKETTE. (NUR PASCAL 2.0)    *)
     VAR KOMMANDO: TEXT;
```

Bild 79: Natürliches Mischsortieren (Fortsetzung nächste Seite) ►

```
 BEGIN
   OPEN(KOMMANDO,8,15,'S' + N); CLOSE(KOMMANDO)
 END; (* ERASE *)

PROCEDURE LIST(VAR F: TAPE; N: STRING);
(* ZEIGE DEN INHALT VON FILE F AN                          *)
BEGIN
  WRITELN('INHALT DES FILES ', N, ':');
  OPEN(F,8,3, N + RD); GET(F);          (* NUR PASCAL 2.0 *)
  WHILE NOT EOF(F) DO
    BEGIN  WRITE(F↑.KEY:4); GET(F) END;
  WRITELN;
  CLOSE(F);                             (* NUR PASCAL 2.0 *)
END; (* LIST *)

PROCEDURE NATURALMERGE;
(* SORTIERE FILE C. BENUTZT ZWEI HILFSFILES A UND B      *)
  VAR L  : INTEGER;  (* ANZAHL DER LAEUFE AUF C          *)
      EOR: BOOLEAN;  (* END OF RUN, ENDE DES LAUFS       *)
      A,B: TAPE;     (* HILFSFILES                       *)

  PROCEDURE COPY (VAR X,Y: TAPE);
  (* KOPIERE KOMPONENTE VON X NACH Y, AKTUALISIERE EOR  *)
    VAR BUF: ITEM;
  BEGIN
    BUF.KEY:=X↑.KEY; GET(X);
    Y↑.KEY:=BUF.KEY; PUT(Y);
    IF EOF(X) THEN EOR:= TRUE
              ELSE EOR:= BUF.KEY>X↑.KEY
  END;(* COPY *)

  PROCEDURE COPYRUN(VAR X,Y: TAPE);
  (* KOPIERE LAUF VON X NACH Y                           *)
  BEGIN
    REPEAT COPY(X,Y) UNTIL EOR
  END;(* COPYRUN *)

  PROCEDURE DISTRIBUTE;
  (* KOPIERE LAUEFE VON C ABWECHSELND AUF A UND B        *)
  BEGIN
    REPEAT

    COPYRUN(C,A);
    IF NOT EOF(C) THEN COPYRUN(C,B)
  UNTIL EOF(C)
END; (* DISTRIBUTE *)

PROCEDURE MERGE;
(* MISCHE FILE A UND B ZU FILE C                        *)

  PROCEDURE MERGERUN;
  (* MISCHE LAEUFE VON A UND B ZU LAEUFEN AUF C         *)
  BEGIN
    REPEAT
      IF A↑.KEY<B↑.KEY THEN
       BEGIN COPY(A,C);
         IF EOR THEN COPYRUN(B,C)
       END
      ELSE
       BEGIN COPY(B,C);
         IF EOR THEN COPYRUN(A,C)
       END
    UNTIL EOR
  END; (* MERGERUN *)
```

Bild 79: Natürliches Mischsortieren

```
    BEGIN (* MERGE *)
      REPEAT
        MERGERUN; L:=L+1
      UNTIL EOF(A) OR EOF(B);
      WHILE NOT EOF(A) DO
        BEGIN
          COPYRUN(A,C); L:=L+1
        END;
      WHILE NOT EOF(B) DO
        BEGIN
          COPYRUN(B,C); L:=L+1
        END
    END;(* MERGE *)

  BEGIN (* NATURALMERGE *)
    REPEAT
      ERASE(NAMEA); ERASE(NAMEB);          (* NUR PASCAL 2.0 *)
      OPEN(A,8,3,NAMEA + WR);
      OPEN(B,8,4,NAMEB + WR);
      OPEN(C,8,5,NAMEC + RD); GET(C);
      DISTRIBUTE;
      CLOSE(A); CLOSE(B); CLOSE(C);

      ERASE(NAMEC);                        (* NUR PASCAL 2.0 *)
      OPEN(A,8,3,NAMEA + RD); GET(A);
      OPEN(B,8,4,NAMEB + RD); GET(B);
      OPEN(C,8,5,NAMEC + WR);
      L:=0; MERGE;
      CLOSE(A); CLOSE(B); CLOSE(C);        (* NUR PASCAL 2.0 *)
      LIST(C, NAMEC);
    UNTIL L=1;
  END; (* NATURALMERGE *)

BEGIN (* HAUPTPROGRAMM *)
 WRITELN('SORTIEREN EINES SEQUENTIELLEN FILES:');
 WRITELN('EINGABEZAHLEN: (0 AM ENDE)');
 ERASE(NAMEC);                             (* NUR PASCAL 2.0 *)
 OPEN(C,8,5,NAMEC + WR);

 READ(C↑.KEY);
 REPEAT
   PUT(C);
   READ(C↑.KEY)
 UNTIL C↑.KEY=0;
 CLOSE(C);                                 (* NUR PASCAL 2.0 *)
 LIST(C, NAMEC);
 NATURALMERGE;
END.
```

Bild 79: Natürliches Mischsortieren (Schluß)

Analyse eines Pascal-Programms

Am Beispiel des Programmes in Bild 79 sollen Sie auch lernen, wie man ein fremdes Pascal-Programm *liest*: Dabei beginnt man am besten am **Ende** des Listings. Dort steht das Hauptprogramm, das alle Prozeduren aufruft. In diesem Fall wird dort zunächst das File C mit Werten gefüllt, die von der Tastatur eingelesen werden. Ein wichtiges Detail sind auch die Bedingungen, die REPEAT- und WHILE-Schleifen kontrollieren. Hier wird die Schleife beendet, falls eine 0 von der Tastatur gelesen wurde. Anschließend wird die Prozedur LIST mit dem File C als Parameter aufgerufen. Bei solchen Prozeduraufrufen haben Sie zwei verschiedene Möglichkeiten, die Funktion

des Programmes weiter zu analysieren: Entweder versuchen Sie, die Funktion der Prozedur LIST zu entschlüsseln, oder Sie schließen aus dem Namen der Prozedur und dem Kontext auf die Bedeutung des Unterprogrammes.
An dieser Stelle ist es sicherlich einsichtig, daß die Prozedur LIST den Inhalt des Files am Bildschirm auflistet. An diesen Prozeduraufruf schließt sich die eigentliche Sortieroperation (NATURALMERGE) an. Diese Prozedur können Sie wie das Hauptprogramm in einzelne Teilschritte zerlegen.
NATURALMERGE vollzieht die oben beschriebenen Durchläufe in zwei Schritten. Wieder ist die Bedingung der REPEAT-Schleife (»UNTIL L=1«) entscheidend. Offensichtlich wird die Variable L in der Prozedur MERGE verändert. Ein Blick auf die Variablendeklaration von L (im Block NATURALMERGE) wird Ihnen jetzt etwas weiterhelfen. Den Rest des Programmes sollten Sie zur Übung selbständig weiter analysieren.

Definition eines Laufes

Als eine kleine Hilfe sei noch der Begriff eines **Laufes** (engl. *run*) erläutert. Ein Lauf ist eine geordnete Teilsequenz in einem File. So enthält die Folge

```
(1  4  3  2  7  5  6  8)
```

diese vier Läufe:

```
(1  4)
(3)
(2  7)
(5  6  8)
```

Beispiel für das Programm in Bild 79

Das Programm MERGE liefert für die obige Zahlenfolge im File C in zwei Durchläufen ein sortiertes File:

```
C = (1  4  3  2  7  5  6  8)        verteile C auf A und B
  A = (1  4  2  7)
  B = (3  5  6  8)                  mische A und B zu C
C = (1  3  4  5  6  8  2  7)        verteile C auf A und B
  A = (1  3  4  5  6  8)
  B = (2  7)                        mische A und B zu C
C = (1  2  3  4  5  6  7  8)
```

Natürlich können Sie mit dem Algorithmus nicht nur Dateien mit ganzen Zahlen sortieren. Eine Anpassung des Programmes an beliebige Typen erfolgt im Typdeklarationsteil bei den Typen ITEM und KEY.

Files in Datenstrukturen

Eine Erweiterung des obigen Programmes zeigt auch den Sinn zusammengesetzter Datentypen, die Files als Unterstrukturen enthalten. Im sogenannten *N-Weg-Mischen* werden statt der zwei Hilfsfiles (A und B) insgesamt N verschiedene Files vom Typ TAPE benutzt. Um auf jedes File mit einem Index zuzugreifen, kann man zum Beispiel die folgende Deklaration eines Arrays von Files benutzen:

```
CONST N = 4;                (* Anzahl der Hilfsdateien *)
TYPE  TAPENUMMER = 1..N;
      TAPE = FILE OF INTEGER;
VAR   F       :  ARRAY [TAPENUMMER] OF TAPE;
      F0      : TAPE;
      COMMAND: FILE OF CHAR;
      I,J     : TAPENUMMER;
      L       : INTEGER;    (* ANZAHL DER LÄUFE *);
```

Verteilung von Daten auf N Files

Verwendet man diese Variablenvereinbarungen, so läßt sich die Verteilung (*distribute*) des Files F0 auf F[1] bis F[N] folgendermaßen programmieren:

```
FOR I:= 1 TO N DO
  BEGIN
    OPEN(COMMAND,'SO:HELP' + CHR(I+64)); CLOSE(COMMAND);
    OPEN(F[I],8,2+I,'HELP' + CHR(I+64) + ',SEQ,WRITE')
  END;
OPEN(F0,8,2,'SORT,SEQ,READ'); GET(F0);

J:= N; L:=0;
REPEAT
  IF J = N THEN J:=1
           ELSE J:= J+1;
  (* kopiere einen Lauf von F0 auf das Band j *)
  L:= L+1;
  REPEAT
    F[J]↑:= F0↑; PUT(F[J]); GET(F0);
  UNTIL (F[J]↑ > F0↑) OR EOF(F0);
UNTIL EOF(F0);

CLOSE(F0);
FOR I:= 1 TO N DO CLOSE(F[I]);
```

Bild 80: Verteilung eines Files auf N Files

Operationen auf mehrere Files gleichzeitig

Zunächst wird der Inhalt der N Files F[I] gelöscht. Anschließend werden die Files zum Schreiben eröffnet. Um die 4 Dateien unterscheiden zu können, werden sie folgendermaßen benannt:

HELPA
HELPB
HELPC
HELPD

Außerdem erhalten sie mit dem Ausdruck 2 + I eindeutige Sekundäradressen (3, 4, 5 und 6). Nachdem das File F0 zum Lesen eröffnet wurde, werden

anschließend so lange Läufe von F0 reihum auf F[1], F[2], F[3], F[4], F[1], F[2], ... verteilt, bis das Dateiende von F0 erreicht wurde (EOF(F0) = TRUE).
Durch die Verwendung der OPEN- und CLOSE-Routinen wird zwar ein Programm in *Pascal 2.0* etwas länger als im *report* definiert, jedoch können Sie gezielt (wie in BASIC) auf die Peripheriegeräte des C128 (Floppy, Datasette, Plotter, Modems etc.) zugreifen, da das Konzept der Geräteadressen und Sekundäradressen voll unterstützt wird. Die Wirkung der vordefinierten File-Prozeduren und Funktionen wird in der Dokumentation (Abschnitt 4.4.5) exakt beschrieben. Im dritten Kapitel werden zusätzlich konkrete Beispielprogramme aufgeführt.

Aufgaben

1. Ändern Sie das Programm MERGE in Bild 79, so daß ein File mit Namen und Geburtsdatum sortiert wird:

```
TYPE ITEM = RECORD
              KEY  : STRING[20];
              G    : RECORD
                       T: 1..31;
                       M: 1..12;
                       J: INTEGER
                     END;
            END;
```

2. Kombinieren Sie die Algorithmen der Programme MERGE und QUICKSORT: Vereinbaren Sie ein Array konstanter Größe (z.B. N Elemente) im Hauptspeicher. In diesem Array nehmen Sie eine Vorsortierung des Files C in sortierte Teilsequenzen der Länge N vor. Anschließend können Sie diese Sequenzen durch *natürliches Mischsortieren* zu einem einzigen sortierten File C zusammenfassen.
3. Programmieren Sie das skizzierte N-Weg-Sortieren. Beachten Sie dabei die Hinweise zu Bild 80 bezüglich der Vergabe eindeutiger Namen und Sekundäradressen!

Fortschreiben eines Bestandsfiles mit einem Bewegungsfile

4. Um große Datenbestände, die auf sequentiellen Files gespeichert werden müssen, zu erweitern und zu modifizieren, verwendet man in der kaufmännischen Datenverarbeitung folgendes Verfahren:
   Eine nach einem Schlüssel (z.B. Kontonummer) sortierte **Bestandsdatei** wird mit einer nach demselben Schlüssel sortierten **Bewegungsdatei** zu einer (ebenfalls sortierten) neuen Bestandsdatei *fortgeschrieben*. Man muß also alle Änderungen, die man am Bestand vornehmen will, in einer Bewegungsdatei sammeln:

```
TYPE STAMMRECORD = RECORD
                     KNUMMER    : INTEGER;
                     NAME       : STRING[30];
                     ...
                     KONTOSTAND: REAL
                   END;
     BEWEGUNG    = RECORD
                     KNUMMER    : INTEGER;
                     UMSATZART  : (EIN, AUS,
                                   KONTOAUFGABE);
                     WERT       : REAL
                   END;
VAR ALT, NEU : FILE OF STAMMRECORD;
    BEW      : FILE OF BEWEGUNG;
```

Während man von der Datei ALT Daten nach NEU kopiert, prüft man, ob für die gerade bearbeitete Kontonummer Umsätze im File BEW vorliegen. Diese werden dann mit dem Kontostand verbucht.
Schreiben Sie ein solches Fortschreibungsprogramm, das auch mehrere Transaktionen in der Bewegungsdatei pro Konto erlaubt. Wenn es Sie mehr motiviert, können Sie mit diesem Verfahren auch Adreß-, Schallplatten- und Literaturdateien verwalten.

2.17 Textfiles

Textfiles enthalten Zeichen als Komponenten

Die im letzten Abschnitt vorgestellten Files besitzen Komponenten eines beliebigen skalaren oder zusammengesetzten Typs. Dadurch können effizient und kompakt alle Werte der Komponenten auf einem Hintergrundspeicher dargestellt werden. Jedoch werden die Werte in codierter Form (als Bytefolgen) gespeichert. Diese Codierung der Daten ist eine für den Menschen oder Programme in anderen Programmiersprachen ungeeignete Darstellungsform. Da zur Ein- und Ausgabe bevorzugt **Zeichenfolgen**, wie

```
Dies ist ein Beispiel für
einen Text, der sich
über drei Zeilen erstreckt.
```

verwendet werden, spielen Files mit dem Komponententyp CHAR eine besondere Rolle. In Pascal existiert deshalb ein vordefinierter Typbezeichner:

Deklaration

```
TYPE TEXT = FILE OF CHAR;
```

Vordefinierte Standardfiles

Files des Typs TEXT bestehen also aus einer (beliebig) langen Folge von Zeichen. Sie kennen bereits zwei File-Variablen, die den Typ TEXT besitzen. Es sind dies die im Programmkopf genannten Namen INPUT und OUTPUT, welche folgendermaßen vordefiniert sind:

```
VAR INPUT, OUTPUT: TEXT;
```

Ein- und Ausgaben über Textfiles

In Abschnitt 2.2 wurde erklärt, daß INPUT und OUTPUT die Standardeingabe von der Tastatur bzw. die Standardausgabe an den Bildschirm symbolisieren. Eingaben von der Tastatur und Ausgaben an den Bildschirm stellen nichts anderes als Folgen von Zeichen des Typs CHAR dar, die nur sequentiell lesend oder schreibend bearbeitet werden können.

Aber auch alle anderen Peripheriegeräte am C128 kommunizieren mit dem Computer über Zeichenfolgen, die vom Drucker gedruckt, auf der Floppy gespeichert und von einem Modem eingelesen werden. Zwar könnte man eine zeichenweise Aus- und Eingabe auch auf Textfiles mit den Standardfile-Operationen PUT und GET vornehmen, jedoch existieren einige spezielle Prozeduren und Funktionen in Pascal, mit denen komplette Zeichen**folgen** bearbeitet werden können:

READ(LN) und WRITE(LN) für Textfiles

In Abschnitt 2.5 wurde nämlich nur eine Kurzform der Prozeduren READ(LN) und WRITE(LN) vorgestellt, in der alle Ausgaben am Bildschirm und alle Eingaben von der Tastatur erfolgen. In der allgemeinen Form dieser Prozeduren muß als erster Parameter in der Parameterliste ein File vom Typ TEXT (also FILE OF CHAR) genannt werden, auf dem die auszugebenden Daten zu speichern sind oder von dem die Eingabedaten stammen:

```
WRITE(Filevariable, Ausdruck, Ausdruck, ..., Ausdruck)
WRITELN(Filevariable)
WRITELN(Filevariable, Ausdruck, Ausdruck, ...., Ausdruck)
```

Die obigen Prozeduren speichern Zeichenfolgen in dem durch *File-Variable* spezifizierten File. Die Ausdrücke können, wie in Abschnitt 2.5 beschrieben, um Formatierungsparameter erweitert werden. Mit der Prozedur WRITELN können Sie den ausgegebenen Text (wie es ebenfalls in Abschnitt 2.5 für den Bildschirm erläutert wurde) in **Zeilen** gliedern.

```
PROGRAM TEXTFILE(INPUT,OUTPUT);
  VAR F: TEXT;
      S: STRING;
BEGIN
  OPEN(F,8,3, 'TEXT,SEQ,WRITE');
  WRITELN(F,'Dies ist ein Beispiel für');
  WRITELN(F,'einen Text, der sich');
  WRITELN(F,'über drei Zeilen erstreckt.');
  CLOSE(F);

  OPEN(F,8,3, 'TEXT,SEQ,READ');
  WHILE NOT EOF(F) DO
```

Bild 81: Zeilenstrukturen in Textfiles

```
    BEGIN
      READLN(F,S); WRITELN(S);
    END;
  CLOSE(F);
END.
```

Bild 81: Zeilenstrukturen in Textfiles (Schluß)

Zeilenstrukturen mit Steuerzeichen

Das Ende einer Zeile wird in Files (und am Bildschirm) mit einem Steuerzeichen markiert. In *Pascal 2.0* ist dies das Zeichen CHR(13) (*carriage return* im ASCII). Jedoch müssen Sie sich nicht um solche Realisierungsdetails kümmern, da spezielle Funktionen und Prozeduren zur Behandlung der Zeilenstruktur in Textfiles existieren.
Ist der erste Parameter einer WRITE-Anweisung keine File-Variable, so wird das Standardfile OUTPUT zur Ausgabe benutzt:

`WRITELN('↑':50 , 'ERROR', ERRNUM:3)` entspricht
`WRITELN(OUTPUT,'↑':50, 'ERROR', ERRNUM:3)`

Wie alle anderen Files auch, müssen Files mit dem Komponententyp CHAR vor Schreiboperationen mit RESET (OPEN in *Pascal 2.0*) zum Schreiben eröffnet werden. In dem in Bild 82 angegebenen Programm wird wieder das Muster für eine zeilenweise Ausgabe verwendet, wie es Ihnen bereits in Abschnitt 2.5 und bei der Ausgabe von Matrizen in Abschnitt 2.9.3 begegnet ist.

Ausgabe einer ASCII-Tabelle

Das Programm gibt eine ASCII-Tabelle (also eine Liste aller darstellbaren Zeichen in der Reihenfolge ihrer Codierung) auf einem Drucker aus. Die Variable I gibt die momentan ausgegebene Zeile an. In der inneren FOR-Anweisung für die Variable J wird der ASCII-Code mit der Schrittweite 15 berechnet. Jede Zeile wird mit WRITELN(D) beendet. Bei der Ausgabe werden die Zeichen mit den Codes zwischen 0 und 31 sowie 127 und 159 nicht gedruckt, da sie am Drucker nur Steuerfunktionen (Schriftartwechsel, Papiervorschub etc.) besitzen.

```
PROGRAM ASCII (INPUT,OUTPUT);
  VAR I,J: 0..15;
      D  : TEXT;          (* Filevariable zur Ausgabe *)
BEGIN
  OPEN(D,4,0); (* Eröffne den Druckerkanal *)
  FOR I:= 0 TO 15 DO
    BEGIN
      FOR J:= 0 TO 15 DO
        IF J IN [0,1,8,9] THEN
```

Bild 82: Ausgabe einer ASCII-Tabelle (Fortsetzung nächste Seite) ►

```
            (* Steuerzeichen auslassen *)
            WRITE(D,' ':3)
          ELSE
            WRITE(D,' ',CHR(I+16*J),' ');
        WRITELN(D);
      END;
    CLOSE(D)
  END.
```

Bild 82: Ausgabe einer ASCII-Tabelle (Schluß)

Umleitung der Ausgabe durch die OPEN-Prozedur

Sollten Sie keinen Drucker besitzen, so können Sie die Ausgabe auch auf einem anderen Peripheriegerät vornehmen. Dazu müssen Sie nur im OPEN-Befehl den Geräteparameter (4 für Druckerausgabe) ändern. So besitzt z.B. der Bildschirm die Geräteadresse 3. In diesem Buch können natürlich nicht alle Eigenheiten des Betriebssystems des C128 besprochen werden. Solche Informationen finden Sie einerseits im BASIC-Handbuch und andererseits bei den Bedienungsanleitungen der Geräte (Drucker, Floppy). Dort wird auch die Bedeutung der Sekundäradresse (0) für jedes Gerät erläutert.

```
OPEN(D,4,7)            Beim Drucker MPS-802 Ausgabe in Kleinschrift
OPEN(D,3,0)            Ausgabe auf den Bildschirm
OPEN(D,1,2,'ASCII')    Ausgabe auf ein Kassetten-File mit dem Namen
                       ASCII
OPEN(D,8,3,'ASCII,SEQ,WRITE')
```

Die letzte Angabe erzeugt eine sequentielle Datei mit dem Namen »ASCII« auf der Diskette mit der Geräteadresse 8. Solche Textdateien können Sie übrigens mit dem Editor des Pascal-Systems lesen und schreiben (*primary-commands* INPUT und OUTPUT siehe 4.2.10 und 4.2.11).

Umleitung der Standard-Textfiles INPUT und OUTPUT

Eine sehr nützliche Möglichkeit von *Pascal 2.0* besteht darin, die Standardausgabe (OUTPUT) auf ein anderes Gerät als den Bildschirm umzuleiten. Das folgende kleine Programm druckt eine gedämpfte Schwingung auf dem Bildschirm. Bei Bedarf können Sie die Ausgabe jedoch mit dem Prozeduraufruf OPEN(OUTPUT,4,0) auf den Drucker umlenken. Um wieder zur Bildschirmausgabe zurückzuschalten, genügt der Prozeduraufruf CLOSE(OUTPUT).

```
PROGRAM SIMPLECURVE (INPUT,OUTPUT);
  CONST D   = 0.00624;
        S   = 32;
        H   = 34;
        C   = 6.28318; (* 2 * PI *)
        LIM = 32;
```

Bild 83: Umleitung der Standardausgabe

```
  VAR X,Y: REAL;
        I: INTEGER;
BEGIN
  WRITE('Ausgabe auf den Drucker? N' #157); READLN(C);
  IF C<>'N' THEN
    OPEN(OUTPUT,4,0); (* Umleitung auf den Drucker *)
  FOR I:= 0 TO LIM DO
    BEGIN
      X:= D*I; Y:= EXP(-X)* SIN(C*X);
      N:= ROUND(S*Y) + H;
      WRITE('*':N);
    END;
  CLOSE(OUTPUT);  (* OUTPUT wieder zum Bildschirm *)
END.
```

Bild 83: Umleitung der Standardausgabe (Schluß)

Eingabe von Textfiles mit READ(LN)

Natürlich gelten analog die Prozeduren READ und READLN nicht nur für das Standard-Eingabe-Textfile INPUT. Die allgemeine Syntax der Eingabeoperationen lautet nämlich:

```
READ(Filevariable, Variable, Variable, ..., Variable)
READLN(Filevariable)
READLN(Filevariable, Variable, Variable, ..., Variable)
```

Die Eingabe erfolgt wie in Abschnitt 2.5 zeilenweise. Damit lassen sich also Werte der Typen CHAR, INTEGER, REAL und STRING einlesen. Die Regeln bezüglich der Syntax der eingelesenen Zahlen wurden bereits in Abschnitt 2.5 beschrieben. Wird bei den Prozeduren als erster Parameter keine File-Variable angegeben, so erfolgt die Eingabe vom Standard-Eingabemedium INPUT:

```
READ(A,X,S)              entspricht
READ(INPUT,A,X,S)
```

Die Funktion EOLN erkennt Zeilenwechsel

Sowohl bei der Eingabe von der Tastatur als auch bei jedem anderen Textfile besteht die Möglichkeit, zu prüfen, ob bei der Eingabe das Ende einer Zeile erreicht wurde. Die Funktion

```
EOLN(Filevariable)
```

(end of line, Zeilenende) liefert ein boolesches Ergebnis. Wurde beim Einlesen einer Zahl oder eines Zeichens vom File *Filevariable* das Zeilenende erreicht, so ist EOLN(*Filevariable*) = TRUE.

```
PROGRAM EOLTEST(INPUT,OUTPUT);
  VAR I,ZAHL: INTEGER;
BEGIN
  WRITELN('Geben Sie eine Folge von 5 Zahlen ');
  WRITELN('in verschiedenen Zeilen ein:');
  FOR I:= 1 TO 5 DO
    BEGIN
      READ(ZAHL);
      IF EOLN(INPUT) THEN
        WRITE('Zeilenende nach', ZAHL);
    END;
END.
```

Bild 84: Erkennen von Zeilenwechseln

In Bild 84 wurde die Funktion EOLN(INPUT) verwendet, um das Ende einer Eingabezeile beim File INPUT (Benutzer betätigt die Return-Taste) zu erkennen. Wie bei READ(LN) kann auch bei EOLN die Angabe INPUT entfallen, so daß man in Bild 84 die IF-Anweisung abkürzen könnte:

```
IF EOLN THEN
  WRITE('Zeilenende nach', ZAHL);
```

READLN erzwingt Zeilenwechsel beim Lesen von Textfiles

Die Prozedur READLN(F) läßt sich formal durch folgende Anweisungsfolge definieren (die Variable CH ist vom Typ CHAR):

```
READLN(F)                          entspricht
WHILE NOT EOLN(F) DO READ(F,CH)
```

Das folgende Programm (Bild 85) demonstriert die Eingabe von Files mit READ. Das sequentielle File mit dem Namen »DATA« auf der Diskette beinhaltet in jeder Zeile mehrere reelle Zahlen. Die Zeilensumme über diese Zahlen soll am Bildschirm angezeigt werden. Hat das File folgenden Inhalt

```
3.142     22          -0.345 0.33
1 2 3 4
            3
-8   -8                  -8
```

soll also die Ausgabe

```
25.127 10 3 -24
```

Beispiel: Zeilensummen-Berechnung

erzeugt werden. Da die Länge einer Zeile und die Länge des Files nicht *à priori* bekannt sind, werden WHILE- und REPEAT-Anweisungen mit den Funktionen EOLN(DATEN) und EOF(DATEN) als Abbruchkriterien verwendet.

```
PROGRAM ZEILENSUMME (INPUT, OUTPUT);
  VAR DATEN: TEXT;

  PROCEDURE ADD(VAR F: TEXT);
  (* ADDIERE DIE WERTE IN DEN ZEILEN VON F    *)
  (* DRUCKE DIE SUMMEN                        *)
    VAR R, SIGMA: REAL;
  BEGIN
    WHILE NOT EOF(F) DO
      BEGIN SIGMA:=0.0;
        REPEAT
          READ(F,R); SIGMA:= SIGMA + R;
        UNTIL EOLN(F);
        WRITE(SIGMA:10:3);
      END;
  END; (* ADD *)

BEGIN
  OPEN(DATEN, 8, 3, 'DATA,SEQ,READ');
  ADD(DATEN); WRITELN;
  CLOSE(DATEN)
END.
```

Bild 85: Schleifen mit EOLN und EOF

Bemerkenswert ist vielleicht noch die Übergabe von File-Variablen als Parameter an Funktionen und Prozeduren, da in diesem Fall Files nur als *Variablenparameter* übergeben werden dürfen. Die Prozedur ADD hätte also nicht folgendermaßen vereinbart werden dürfen:

Falsch!

```
PROCEDURE ADD(F: TEXT);
```

Kopieren eines Files zeichenweise

Das Programm in Bild 86 zeigt abschließend, wie man eine Textdatei zeichenweise bearbeitet und gleichzeitig ein File der gleichen Zeilenstruktur erzeugt. Es wird eine Datei gelesen und alle Kleinbuchstaben oder Grafikzeichen (mit Ordinalwerten ORD(CH) >=128) in Großbuchstaben umgewandelt.

```
PROGRAM KONVERT (INPUT, OUTPUT);
  VAR EINGABE,AUSGABE: TEXT;
      NAME           : STRING[16];
      CH             : CHAR;
```

Bild 86: Umwandlung einer Datei in Großbuchstaben (Fortsetzung nächste Seite) ►

```
BEGIN
  WRITE('FILENAMEN :'); READLN(NAME);
  OPEN(EINGABE, 8, 3, NAME + ',SEQ,READ');
  OPEN(AUSGABE, 8, 4, NAME + '.G,SEQ,WRITE');
  WHILE NOT EOF(EINGABE) DO
  BEGIN
    READ(EINGABE, CH);
    WHILE NOT EOLN(EINGABE) DO
      BEGIN
        IF ORD(CH) > 127 THEN CH:= CHR(ORD(CH)-128);
        WRITE(AUSGABE,CH);
        READ(EINGABE,CH);
      END;
    WRITELN(AUSGABE);
  END;
  CLOSE(EINGABE); CLOSE(AUSGABE);
END.
```

Bild 86: Umwandlung einer Datei in Großbuchstaben (Schluß)

Natürlich können Sie das Programm auch mit Files auf anderen Speichermedien testen, indem Sie die Parameter bei OPEN geeignet wählen.

Kopieren eines Files zeilenweise

Da jeder Aufruf der Standardprozeduren READ und WRITE neben der eigentlichen Zeichenein- und ausgabe zunächst das Peripheriegerät adressieren muß, kann man die Verarbeitungsgeschwindigkeit des Programmes dadurch erhöhen, daß man jeweils vollständige Zeilen (Strings) einliest und ausgibt und diese im Hauptspeicher bearbeitet:

```
PROGRAM STRINGKONVERT (INPUT, OUTPUT);
  VAR EINGABE,AUSGABE: TEXT;
      NAME           : STRING[16];
      I              : INTEGER;
      LINE           : STRING[255];
BEGIN
  WRITE('FILENAMEN :'); READLN(NAME);
  OPEN(EINGABE, 8, 3, NAME + ',SEQ,READ');
  OPEN(AUSGABE, 8, 4, NAME + '.G,SEQ,WRITE');
  WHILE NOT EOF(EINGABE) DO
    BEGIN
      READLN(EINGABE, LINE);
      FOR I:= 1 TO LENGTH(LINE) DO
```

Bild 87: Umwandlung mit Zeilenpuffer

```
      IF ORD(LINE[I]) > 127 THEN
        LINE[I]:= CHR(ORD(LINE[I])-128);
    WRITELN(AUSGABE, LINE);
  END;
  CLOSE(EINGABE); CLOSE(AUSGABE);
END.
```

Bild 87: Umwandlung mit Zeilenpuffer (Schluß)

Unterschied zwischen Files und Textfiles

In aller Deutlichkeit soll nochmals auf die Unterschiede zwischen der Speicherung von Werten in Textfiles und »normalen« Files hingeweisen werden. Möchten Sie z.B. 100 Zahlen des Typs REAL speichern, so könnten Sie folgende Programmstücke verwenden:

```
PROGRAM WRITEREAL1 (INPUT,OUTPUT);
  VAR REALS: FILE OF REAL;
      I    : INTEGER;
BEGIN
  OPEN(REALS,8,3,'ZAHLEN,SEQ,WRITE');
  FOR I:= 1 TO 100 DO
    BEGIN REALS↑:= 1 / I; PUT(REALS); END;
  CLOSE(REALS);
END.

PROGRAM WRITEREAL2 (INPUT,OUTPUT);
  VAR T    : TEXT;
      I    : INTEGER;
BEGIN
  OPEN(T,8,3,'ZAHLEN,SEQ,WRITE');
  FOR I:= 1 TO 100 DO
    WRITE(T, 1/I:15);
  CLOSE(T);
END.
```

Bild 88: Speicherung reeller Zahlen

1. Bei der Verwendung eines FILE OF REAL werden die reellen Zahlen so gespeichert, wie sie im Speicher des Computers codiert sind (5 Byte pro Zahl). Daher erfolgt die Ausgabe extrem schnell und die Daten sind sehr kompakt gespeichert.
2. Bei einem File des Typs TEXT muß bei jeder Ausgabe eine reelle Zahl in eine formatierte Folge von Zeichen umgewandelt werden (»0.10000000E-01«). Dies erfordert einen nicht unerheblichen Aufwand.

Jedoch können die so gespeicherten Daten z.B. auf einen Drucker ausgegeben werden und von einem Menschen gelesen werden. In der Sprache BASIC werden ebenfalls alle Zahlen als Zeichenfolgen gespeichert. Möchten Sie also die in Pascal erstellten Daten in BASIC weiterverwenden, so müssen Sie das Schema aus Programm WRITEREAL2 verwenden.

Aufgaben

Seitenformatierung am Drucker

1. Erstellen Sie ein Druckprogramm, das den Inhalt eines Daten-Files, wie es zum Beispiel in Aufgabe 1 in Abschnitt 2.16 beschrieben wurde, formatiert als Liste ausgibt. Finden Sie ein möglichst allgemein verwendbares Verfahren, um jede Seite mit einem Listenkopf (Titel und Seitennummer) zu versehen.

Einstufige Gruppenkontrolle

2. Gegeben ist ein Daten-File, das die Umsätze von Vertretern im Bundesgebiet für ein Jahr enthält. Dieses File ist nach dem Feld *Postleitzahl* aufsteigend sortiert. Drucken Sie eine Liste, die alle Umsätze im Bundesgebiet enthält. Außerdem sollen Zwischensummen gebildet werden, aus denen die Gesamtumsätze in jedem PLZ-Bereich (also z.B. 6000–6999, 7000–7999 etc.) hervorgehen:

PLZ	Stadt	Vertreter	Monat	Umsatz
6000	Frankfurt/M.	Müller	3	323000.00
6200	Wiesbaden	Meier	4	3000.00
		Mogel	12	100000.00
6991	Wildentierbach	Schulze	12	3.00
			SUMME:	426003.00
7000	Stuttgart	Müller	6	4500.00
			SUMME:	4500.00

Mehrstufige Gruppenkontrolle

3. Eine solche *Gruppenkontrolle* wie in Aufgabe 2 läßt sich auch mehrstufig anwenden. Innerhalb jedes PLZ-Gebietes könnte man (bei einer entsprechenden Sortierung der Ausgangsdaten) auch eine zusätzliche Aufschlüsselung nach Monatsumsätzen vornehmen. Im Programm muß also ein Vergleich des laufenden mit dem nachfolgenden (Teil-)Schlüssel vorgenommen werden. Das generelle Schema einer zweistufigen Gruppenkontrolle ist also folgendes:

```
Dateien eröffnen
Ersten Satz lesen
WHILE NOT Dateiende erreicht DO
BEGIN
```

```
  Vorlauf Stufe 2
  REPEAT
    Vorlauf Stufe 1
     REPEAT
      Bearbeite Einzelposten
      Neuen Satz lesen
    UNTIL Wechsel 1
    Gruppenabschluß 1
  UNTIL Wechsel 2
  Gruppenabschluß 2
END
Dateien schließen
```

Wechsel 1 bezeichnet im oben skizzierten Beispiel einen Wechsel des Monats, während Wechsel 2 eine Änderung des übergeordneten Gruppenkriteriums (PLZ-Bereich) bedeutet. Wechsel 1 muß natürlich auch durch ein Dateiende und Wechsel 2 hervorgerufen werden. Ein Wechsel 2 wird auch durch das Dateiende provoziert.
Der Vorlauf für eine Gruppe enthält das Löschen von Summenfeldern, den Druck von Überschriften etc., während der Gruppenabschluß z.B. den Druck einer Summenzeile unter einer Gruppe bedeutet.

2.18 Dynamische Datenstrukturen

Definition: statische Datenstrukturen

Mit Ausnahme der Variablen vom Typ File sind alle bisher vorgestellten Strukturen (Arrays, Records, Mengen) statisch. Das heißt, sie behalten während ihrer Gültigkeit die Struktur bei, die bei der Deklaration vereinbart wurde. Die Gültigkeit dieser Variablen reicht vom Eintritt in einen Block (Hauptprogramm, Prozedur oder Funktion) bis zum Verlassen dieses Blockes (Programmende oder Rücksprung aus einer Prozedur).

Definition: dynamische Datenstrukturen

In diesem Abschnitt wird beschrieben, wie man in Pascal Objekte konstruiert, die während der Programmlaufzeit nicht nur wachsen oder schrumpfen, sondern auch **dynamisch** zu Listen und beliebigen Netzen verbunden werden können. Zum Zeitpunkt der Übersetzung wird nur die (statische) Struktur der Elemente definiert. Diese Bausteine besitzen meist die Struktur eines Records. Der Speicherplatz für die verschiedenen Records wird dann zur Programmlaufzeit je nach Bedarf zur Verfügung gestellt. Die Verbindung zu komplexen Strukturen geschieht über **Zeiger**, die von Record zu Record führen.
Das Konzept der Zeigertypen stellt einen großen Fortschritt der Sprache Pascal gegenüber *klassischen* Sprachen wie Fortran, COBOL und BASIC dar. Oft ist jedoch festzustellen, daß auch ambitionierte Einsteiger in Pascal an diesem für sie qualitativ neuen Datentyp scheitern. Tatsächlich stellt die Arbeit mit

dynamischen Datenstrukturen den Programmierer vor völlig neue Probleme bei der Programmentwicklung und -überprüfung. Daher soll in diesem Abschnitt ein eher pragmatischer Einstieg in das Thema gefunden werden.

Eine Kundenliste als dynamische Datenstruktur

Die Aufgabe besteht darin, in einem Unternehmen eine *Liste* von Kunden anzulegen. Von jedem Kunden sollen der Name und die Kundennummer gespeichert werden. Da *a priori* nicht bekannt ist, wie viele Kunden in Zukunft zu speichern sind, kann man zur Speicherung kein Array mit Records verwenden. Andererseits ist die rein sequentielle Zugriffsmethode auf ein (langsames) Disketten-File nicht zu verwenden. Diese Aufgabe ist ein typisches Beispiel für die Anwendung einer *Liste*, die dynamisch durch Zeiger gebildet wird. Die benötigten Strukturen werden im folgenden Deklarationsteil beschrieben:

```
TYPE KUNDENZEIGER  = ↑ KUNDE;
     KUNDE= RECORD
                NAME      : STRING[10];
                KNUMMER   : INTEGER;
                NAECHSTER: KUNDENZEIGER;
            END;
VAR  KUNDE1, KUNDENEU, LETZTERKUNDE: KUNDENZEIGER;
```

Bild 89: Deklaration von Zeigertypen

Deklaration eines Zeigertyps

Mit der Typdeklaration in Bild 89 wird ein Typ KUNDE definiert, der die gewünschten Informationen (NAME, KNUMMER) für jeden Kunden speichert. Außerdem existiert ein Typ KUNDENZEIGER, der als Werte Zeiger (*pointer*) auf solche Kunden-Records besitzt. Als Variablen wurden keine Kunden-Records, sondern nur Zeigervariablen deklariert.

Sie sollten sich einen Zeiger recht plastisch als einen Pfeil (↑) vorstellen, der immer auf einen Wert des zugehörigen Typs zeigt (siehe Bild 90). KUNDENZEIGER weisen also immer auf Records vom Typ KUNDE.

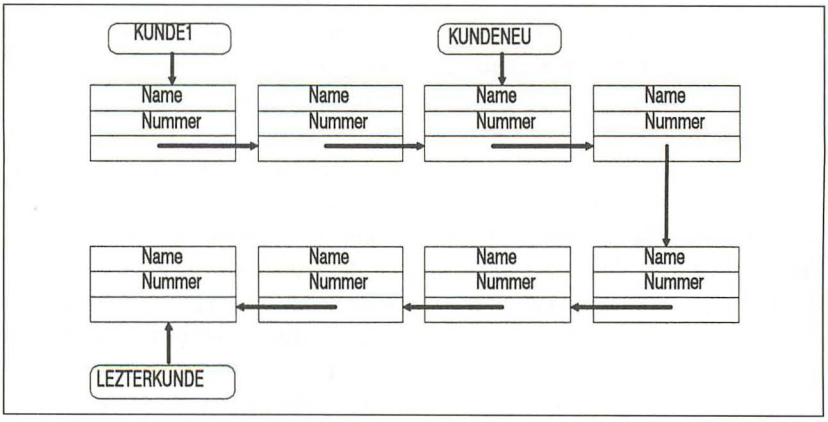


Bild 90: Zeigerstrukturen

Struktur einer Liste

Ordnet man die Zeiger wie in Bild 90, so erhält man eine Kundenliste: Jeder Kunde besitzt einen Zeiger auf seinen Nachfolger in der Liste. Von *außerhalb* verweisen noch die Zeiger KUNDE1, KUNDENEU und LETZTERKUNDE in die Liste. Bild 90 stellt eine *Momentaufnahme* der Listenstruktur dar. Durch *Umhängen* von Zeigern und Records kann man die Liste umsortieren, verlängern und verkürzen.

Schrittweiser Aufbau einer Liste

Zum Aufbau einer (Kunden-)Liste geht man nun folgendermaßen vor: Man erzeugt für jeden neuen Kunden einen Record vom Typ KUNDE. Diese Records werden nun durch Zeiger vom Typ KUNDENZEIGER *verkettet*. Der Zeiger KUNDE1 zeigt dabei immer auf den ersten Kunden-Record in der Liste. Jeder Record enthält im Feld NAECHSTER einen Zeiger auf seinen Nachfolger in der Liste.

Da die Kunden-Records keinen eigenen Namen besitzen, kann man Kunden-Records auch nur durch die Angabe eines Zeigers ansprechen. Man bezeichnet deshalb dynamische Objekte als *anonym*.

Die Standardprozedur NEW

Um Speicherplatz für einen Kunden-Record zur Verfügung zu stellen, benutzt man die Standardprozedur NEW. Sie erzeugt irgendwo im Speicher Platz für einen Record. Um nun auf diesen Record zuzugreifen, verlangt die Prozedur eine Zeigervariable vom Typ KUNDENZEIGER als aktuellen Parameter. Dieser Zeigervariablen wird die Adresse des neuen Records vom Typ KUNDE zugewiesen:

```
NEW(KUNDENEU)
```

Über den Zeiger KUNDENEU kann man jetzt den Record vom Typ KUNDE mit Werten füllen. Da bei den Zuweisungen nicht die Zeigervariable selbst, sondern das Objekt, auf das der Zeiger zeigt, gemeint ist, benutzt man den Pfeil (↑) nach dem Variablennamen:

```
KUNDENEU↑.NAME:='Jones';
KUNDENEU↑.KNUMMER:= 1111;
```

Zugriff auf dynamische Variablen über Dereferenzierung

Da ein Zeiger eine Referenz auf ein dynamisches Objekt darstellt, nennt man den Pfeil auch Dereferenzier-Operator. Um nun einen anderen Zeiger (nämlich KUNDE1) auf den mit NEW erzeugten Kunden-Record zu setzen, führt man eine Zuweisung zwischen Zeigern durch:

```
KUNDE1:= KUNDENEU
```

Damit Sie den Unterschied zwischen Zeigern und den durch sie referenzierten Objekten erkennen, wird noch ein neuer Kunde *hinter* »Jones« an die Liste gehängt. Zunächst muß ein neuer Record vom Typ KUNDE gebildet werden:

```
NEW(LETZTERKUNDE)
```

Dann wird der Inhalt von LETZTERKUNDE↑ initialisiert:

```
LETZTERKUNDE↑.NAME := 'Jackson';
LETZTERKUNDE↑.KNUMMER:= 2222;
```

Verketten von Records über Zeiger

Nun muß »Jackson« als Nachfolger von »Jones« in die Liste gehängt werden. Dazu verwendet man den Zeiger NAECHSTER im Record von »Jones«, auf den ja noch KUNDENEU zeigt. NAECHSTER soll auf den Record von »Jackson« zeigen, der durch LETZTERKUNDE referenziert wird:

```
KUNDENEU↑.NAECHSTER:= LETZTERKUNDE
```

Bisher wurden nur Zuweisungen an Zeigervariablen oder Kunden-Records durchgeführt. Der Inhalt der Records läßt sich jedoch auch in Ausdrücken und Ausgabeanweisungen verwenden. Zum Zugriff benutzt man wieder die Zeigervariable mit dem nachgestellten Dereferenzieroperator (↑).

```
WRITELN(KUNDE1↑.NAME       , KUNDE1↑.KNUMMER);
WRITELN(LETZTERKUNDE↑.NAME, LETZTERKUNDE↑.KNUMMER);
WRITELN(KUNDE1↑.NAECHSTER↑.NAME);
```

Zeiger in Ausdrücken

Die Bedeutung der ersten beiden Ausgaben ist offensichtlich: Die durch KUNDE1 und LETZTERKUNDE referenzierten Records (»Jones« und »Jackson«) werden angezeigt. Interessanter ist die letzte Ausgabeanweisung, da dort zweimal dereferenziert wird: Es wird der Name des Nachfolgers des Records KUNDE1 gedruckt. Dies ist demnach der 2. Kunde (»Jackson«).

Bild 91 zeigt alle obigen Operationen zusammenfassend in Form eines vollständigen Programmes. Das Ergebnis jeder durch Zahlen im Listing markierten Operationen wird grafisch in Bild 92 dargestellt. Ein anschauliches Bild dynamischer Strukturen stellt die einzelnen Records als Kästchen dar, während Zeiger durch Pfeile symbolisiert werden, die von Record zu Record führen.

```
PROGRAM ZEIGER (INPUT, OUTPUT);
  TYPE KUNDENZEIGER  = ↑ KUNDE;
       KUNDE = RECORD
                 NAME      : STRING[10];
                 KNUMMER   : INTEGER;
                 NAECHSTER: KUNDENZEIGER;
               END;
  VAR  KUNDE1, KUNDENEU, LETZTERKUNDE: KUNDENZEIGER;
BEGIN
  NEW(KUNDENEU);                          (* 1. *)
  KUNDENEU↑.NAME:='Jones';
  KUNDENEU↑.KNUMMER:= 1111;               (* 2. *)
  KUNDE1:= KUNDENEU;                      (* 3. *)
  NEW(LETZTERKUNDE);
  LETZTERKUNDE↑.NAME := 'Jackson';
  LETZTERKUNDE↑.KNUMMER:= 2222;          (* 4. *)
  KUNDENEU↑.NAECHSTER:= LETZTERKUNDE;    (* 5. *)
```

Bild 91: Operationen mit Zeigervariablen

```
  WRITELN(KUNDE1↑.NAME, KUNDE1↑.KNUMMER);
  WRITELN(LETZTERKUNDE↑.NAME, LETZTERKUNDE↑.KNUMMER);
  WRITELN(KUNDE1↑.NAECHTSER↑.NAME);
END.
```

Bild 91: Operationen mit Zeigervariablen (Schluß)

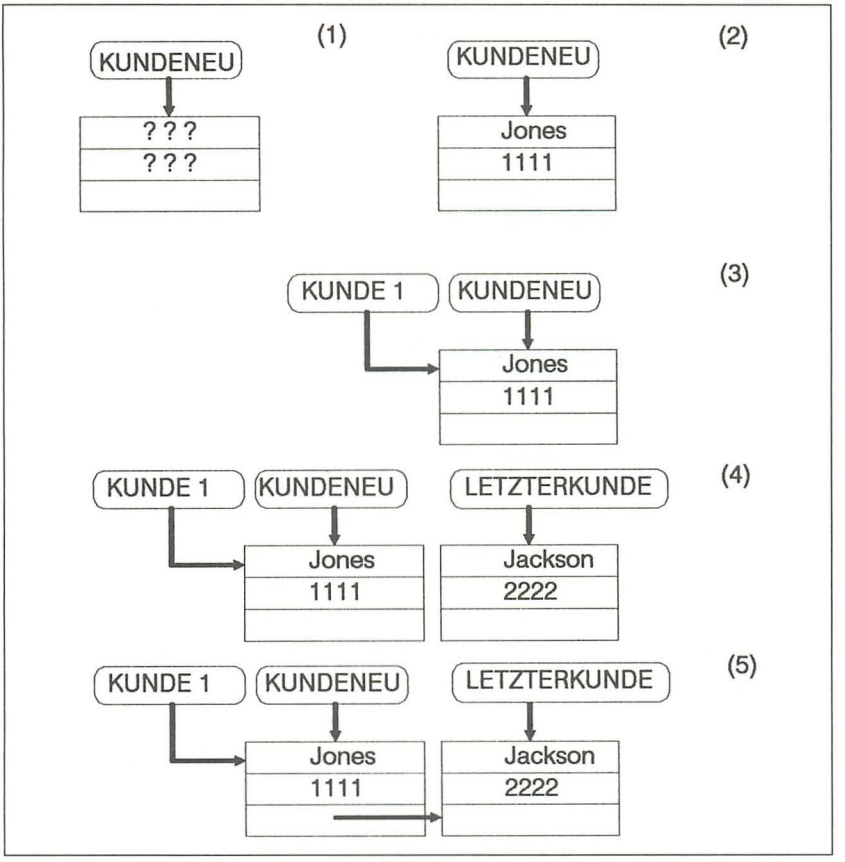


Bild 92: Aufbau der Kundenliste

Ein grundsätzliches Problem wurde noch nicht beachtet: Was passiert mit Zeigern, die (noch) auf **keinen** Record zeigen? So hat z.B. der Record LETZTERKUNDE↑ (»Jackson«) keinen Nachfolger. Eine Möglichkeit besteht darin, jeden Record um ein boolesches Feld zu erweitern, das angibt, ob ein Nachfolger existiert oder nicht. Da dieser Fall bei der Arbeit mit Zeigern ständig auftritt, ist der Wertebereich aller Zeigertypen um den Wert NIL erweitert. Besitzt ein Zeiger P den Wert NIL, so existiert kein Objekt P↑. Deshalb belegt man das Feld NAECHSTER bei dem Record »Jackson« mit NIL:

Der Wert NIL

```
LETZTERKUNDE↑.NAECHSTER:= NIL;
```

Bevor Sie im nächsten Abschnitt typische Datenstrukturen kennenlernen, die mit Zeigern in Pascal realisiert werden, fassen wir die allgemeinen Regeln für die Arbeit mit Zeigern zusammen:

Allgemeine Syntax der Deklaration eines Zeigertyps

- Ein Zeigertyp Z auf Objekte eines beliebigen (strukturierten oder unstrukturierten) Typs T wird deklariert durch
  `TYPE Z = ↑ T;`
- Soll ein Typ, der durch Zeiger angesprochen wird, selbst Zeiger enthalten, so könnten Probleme mit der in Abschnitt 2.15 genannten Regel entstehen, die verlangt, daß jeder Name vor seiner Anwendung deklariert werden muß:

Falsch! (Z ist hier noch unbekannt)

```
TYPE T = RECORD
           ...
           TT : Z;
         END;
 Z = ↑ T
```

Deshalb gibt es von dieser Regel eine Ausnahme: In der Deklaration eines Zeigertyps kann ein Typname verwendet werden, der noch nicht deklariert wurde. Die Deklaration dieses Namens muß jedoch noch im selben Typvereinbarungsteil erfolgen:

Richtig! (T darf hier noch unbekannt sein)

```
TYPE Z = ↑ T;
     T = RECORD
           ...
           TT: Z;
         END;
```

Allgemeine Regeln für Operationen mit Zeigern

- Um ein neues dynamisches Objekt vom Typ T zu erzeugen, ruft man die Prozedur NEW mit einer Variablen vom Typ Z = ↑ T auf.
- Enthält eine Zeigervariable V einen Zeiger auf ein Objekt, das mit NEW erzeugt wurde, so bezeichnet V↑ dieses Objekt.
- Der Wertebereich jeder Zeigervariablen V umfaßt auch den Wert NIL. Ein Zugriff auf das Element V↑ ist für V=NIL nicht zulässig.

Typbindung bei Zeigervariablen

Eine Tatsache muß noch besonders betont werden. Zwar kann ein Zeiger während der Laufzeit auf beliebige Objekte gesetzt werden, jedoch bleibt in jedem Fall die Typbindung von Pascal in Kraft. Dies unterscheidet Zeiger von reinen Adressen in Assembler. Konkret heißt dies, daß eine Zeigervariable, die mit

`VAR V: ↑T;`

deklariert wurde, nur auf Objekte vom Typ T zeigen kann. So ist also die folgende Anweisung nach der angegebenen Deklaration von ZI↑ nicht zulässig:

```
VAR ZI: ↑ INTEGER;
```

Falsch! `ZI↑:= 'A'`

Richtig! `ZI↑:= 312`

2.18.1 Lineare Strukturen (Listen)

Definition: Liste

Im vorangegangenen Abschnitt wurden bereits die ersten Schritte zum Aufbau einer Liste von Kunden-Records gemacht. Dabei lag dem Beispiel eine eher intuitive Vorstellung einer Liste zugrunde, die man bei Bedarf am Ende erweitert.

Grundsätzlich bezeichnet man in Pascal jede lineare Datenstruktur, die durch Zeiger gebildet wird, als eine *Liste*. Linear bedeutet in diesem Zusammenhang, daß jedes Element maximal einen Vorgänger und Nachfolger besitzt.

Folgende Operationen sind mit Listen möglich:

1. Start mit der leeren Liste
2. Erweitern der Liste
3. Löschen in der Liste

Listentypen und ihre Operationen

Es gibt zahlreiche verschiedene Listentypen, die sich durch die Art der Verzeigerung unterscheiden. Wenn Sie noch einmal Bild 90 betrachten, werden Sie feststellen, daß man mit der Operation

```
KUNDENEU:= KUNDENEU↑.NAECHSTER
```

ohne Probleme die Liste *vorwärts* durchlaufen kann. Andererseits ist es (ohne einen Zugriff auf andere Zeiger) nicht möglich, von LETZTERKUNDE zurück zum Vorgänger in der Liste zu gelangen. So bestimmt die Zeigerstruktur die Art der möglichen Zugriffe auf eine Liste.

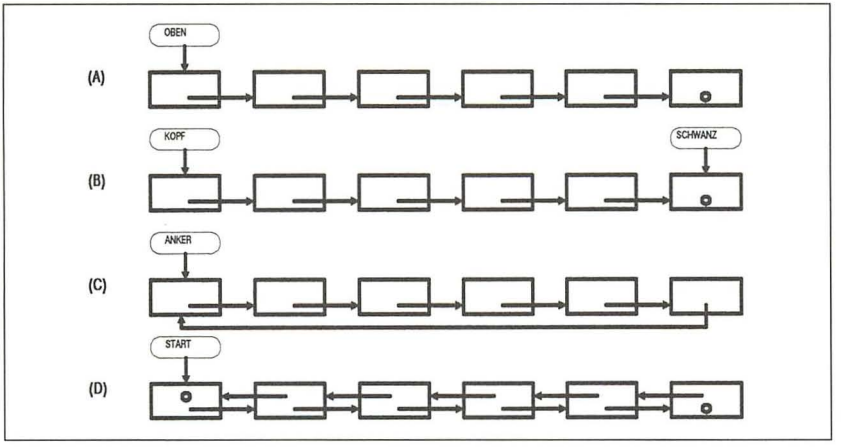


Bild 93: Verschiedene Listen

In Bild 93 sind die wichtigsten Listentypen grafisch dargestellt.

A Kellerspeicher (*stack*, Stapelspeicher)
B Schlange (*queue*)
C Ringspeicher
D Doppelt verkettete Liste

Generell besteht eine Liste (A bis D) aus nur wenigen statischen Zeigervariablen (OBEN, KOPF, SCHWANZ, ANKER, START), die quasi einen

Einstiegspunkt für eine komplexe Struktur mit eventuell vielen hundert dynamischen Datenelementen bilden. Die Elemente erreicht man ausgehend von diesen Einstiegspunkten durch Verfolgung der Zeigerketten.
Bis auf den Ringspeicher wird jeder dieser Typen durch kurze Demonstrationsprogramme vorgestellt, die das Einfügen und Löschen von Namen in einer Liste ermöglichen. Eine etwas erweiterte, einfach verkettete Liste wird anschließend zur Speicherung einer alphabetisch sortierten Kundenliste verwendet.

Kellerspeicher

Bei einem **Kellerspeicher** fügt man Elemente bei OBEN ein und löscht sie auch dort wieder. Weil dadurch das zuletzt eingefügte Element zuerst gelöscht wird, heißt ein Kellerspeicher auch LIFO-Speicher (*last in first out*). Operationen mit einem Kellerspeicher sind besonders einfach zu realisieren.

```
PROGRAM STACK (INPUT,OUTPUT);
   TYPE ZEIGER  = ↑ ELEMENT;
        ELEMENT = RECORD
                    TEXT: STRING[20];
                    NEXT: ZEIGER;
                  END;
  VAR  OBEN: ZEIGER;
       C   : CHAR;

PROCEDURE LEERELISTE;
BEGIN
  OBEN:= NIL;
END; (* LEERELISTE *)

PROCEDURE EINFUEGEN;
  VAR NEU: ZEIGER;
BEGIN
  NEW(NEU);
  WRITE('Text :'); READLN(NEU↑.TEXT);
  NEU↑.NEXT:= OBEN;
  OBEN:= NEU;
END; (* EINFUEGEN *)

PROCEDURE LOESCHEN;
BEGIN
  IF OBEN = NIL THEN
    WRITELN('Die Liste ist leer')
```

Bild 94: Ein Kellerspeicher

```
  ELSE
    BEGIN
      WRITE('Gelöscht wurde:', OBEN↑.TEXT);
      OBEN:= OBEN↑.NEXT;
    END;
END; (* LOESCHEN *)

PROCEDURE ANZEIGEN;
  VAR Z: ZEIGER;
BEGIN
  Z:= OBEN;
  WHILE Z<>NIL DO

    BEGIN
      WRITELN(Z↑.TEXT);
      Z:= Z↑.NEXT;
    END;
END; (* ANZEIGEN *)

BEGIN
  LEERELISTE;
  REPEAT
    ANZEIGEN;
    WRITELN('E infuegen');
    WRITELN('L oeschen');
    WRITELN('B eenden');
    WRITELN;
    WRITE('==>');READLN(C);
    CASE C OF
     'E': EINFUEGEN;
     'L': LOESCHEN;
     ELSE (* nichts *)
    END; (* CASE *)
  UNTIL C='B';
END.
```

Bild 94: Ein Kellerspeicher (Schluß)

Operationen mit einem Kellerspeicher

In Bild 94 ist jede Operation durch eine separate Prozedur realisiert. Sie können wahlweise Daten einfügen und löschen. Die Daten werden offensichtlich in umgekehrter Reihenfolge gespeichert. Eine wichtige Programmiertechnik bei der Arbeit mit Zeigern besteht darin, **immer** den Sonderfall zu beachten, daß ein Zeiger den Wert NIL besitzt: Ist zum Beispiel beim Löschen die

Liste leer, so besitzt der Zeiger OBEN den Wert NIL. In diesem Fall würde die Zuweisung

```
OBEN:= OBEN↑.NEXT
```

einen nicht existierenden Wert OBEN↑ adressieren, was über kurz oder lang zu einem Programmabsturz führen wird.

Schlange

Bei einer **Schlange** (man denke an eine Warteschlange) fügt man Elemente bei dem durch SCHWANZ referenzierten Ende ein und löscht sie bei KOPF. Demnach sind Schlangen FIFO-Speicher (*first in first out*). Während die Operation ANZEIGEN für Schlangen und Kellerspeicher identisch ist, muß bei den Operationen EINFUEGEN und LOESCHEN der Sonderfall beachtet werden, daß die Schlange leer war oder leer wird. Im Falle einer leeren Schlange müssen nämlich beide Zeiger (KOPF und SCHWANZ) den Wert NIL erhalten.

```
PROGRAM QUEUE (INPUT,OUTPUT);
  TYPE ZEIGER  = ↑ ELEMENT;
       ELEMENT = RECORD
                   TEXT: STRING[20];
                    NEXT: ZEIGER;
                 END;
  VAR  KOPF, SCHWANZ: ZEIGER;
       C            : CHAR;

  PROCEDURE LEERELISTE;
  BEGIN
    KOPF   := NIL;
    SCHWANZ:= NIL;
  END; (* LEERELISTE *)

  PROCEDURE EINFUEGEN;
    VAR NEU: ZEIGER;
  BEGIN
    NEW(NEU);
    WRITE('Text :'); READLN(NEU↑.TEXT);
    NEU↑.NEXT:= NIL;
    IF SCHWANZ = NIL THEN (* Schlange war leer *)
      KOPF:= NEU
    ELSE
      SCHWANZ↑.NEXT:= NEU;
    SCHWANZ:= NEU;
  END; (* EINFUEGEN *)
```

Bild 95: Eine Schlange

```
  PROCEDURE LOESCHEN;
  BEGIN
    IF KOPF = NIL THEN
      WRITELN('Die Liste ist leer')
    ELSE
      BEGIN
        WRITE('Gelöscht wurde:', KOPF↑.TEXT);
        KOPF:= KOPF↑.NEXT;
        IF KOPF = NIL THEN  (* Schlange wird leer *)
          SCHWANZ:= NIL;
      END;
  END; (* LOESCHEN *)

  PROCEDURE ANZEIGEN;
    VAR Z: ZEIGER;
  BEGIN
    Z:= KOPF;
    WHILE Z<>NIL DO
      BEGIN
        WRITELN(Z↑.TEXT);
        Z:= Z↑.NEXT;
      END;
  END; (* ANZEIGEN *)

BEGIN
  LEERELISTE;
  REPEAT
    ANZEIGEN;
    WRITELN('E infuegen');
    WRITELN('L oeschen');
    WRITELN('B eenden');
    WRITELN;
    WRITE('==>');READLN(C);
    CASE C OF
     'E': EINFUEGEN;
     'L': LOESCHEN;
     ELSE (* nichts *)
    END; (* CASE *)
  UNTIL C='B';
END.
```

Bild 95: Eine Schlange (Schluß)

Ringspeicher

In einigen Anwendungen sind **Ringspeicher** sinnvoll. Hierbei ist keine vollständige Ordnung auf den Elementen definiert. Jedes Element ist Nachfolger eines anderen, so daß keines der Elemente als »Anfang« oder »Ende« gesondert behandelt werden muß. Die statische Variable ANKER wird nur benötigt, um einen *Einstieg* in die dynamische Struktur zu besitzen. Wäre Anker nämlich nicht vorhanden, so könnte man **keines** der anonymen Objekte über einen Zeiger erreichen!

Doppelte Verkettung ist aufwendig, aber flexibel

Relativ aufwendige Operationen beim Löschen und auch beim Einfügen erfordern die Konstruktion einer doppelt verketteten Liste. Ein wesentlicher Vorteil ist die Tatsache, daß man sich in beiden Richtungen in der Liste bewegen kann. Dafür müssen aber in jedem Record zwei Zeiger vorhanden sein, die bei sehr großen Listen auch einen nennenswerten Speicherplatz verbrauchen können.

```
PROGRAM DOUBLE (INPUT,OUTPUT);
  TYPE ZEIGER  = ↑ ELEMENT;
       ELEMENT = RECORD
                   TEXT: STRING[20];
                   NEXT: ZEIGER;
                   PREV: ZEIGER
                 END;
  VAR  START: ZEIGER;
       C    : CHAR;

  PROCEDURE LEERELISTE;
  BEGIN
    START:= NIL;
  END; (* LEERELISTE *)

  PROCEDURE EINFUEGEN;
    VAR NEU: ZEIGER;
  BEGIN
    NEW(NEU);
    WRITE('Text :'); READLN(NEU↑.TEXT);
    IF START<>NIL THEN
      START↑.PREV:= NEU;
    NEU↑.NEXT:= START;
    NEU↑.PREV:= NIL;
    START:= NEU;
  END; (* EINFUEGEN *)
```

Bild 96: Eine doppelt verkettete Liste

```
  PROCEDURE LOESCHEN;
  BEGIN
    IF START = NIL THEN
      WRITELN('Die Liste ist leer')
    ELSE
      BEGIN
        WRITE('Gelöscht wurde:', START↑.TEXT);
        START:= START↑.NEXT;
        IF START<>NIL THEN
          START↑.PRED:= NIL;
      END;
  END; (* LOESCHEN *)

  PROCEDURE ANZEIGEN;
  (* Zur Demonstration vorwärts und rückwärts *)
    VAR Z,Z1: ZEIGER;
  BEGIN
    WRITELN('VORWAERTS:');
    Z:= START; Z1:= NIL;
    WHILE Z<>NIL DO
      BEGIN
        WRITELN(Z↑.TEXT);
        Z1:= Z; Z:= Z↑.NEXT;
      END;
    WRITELN('RUECKWAERTS:');
    WHILE Z1<>NIL DO
      BEGIN
         WRITELN(Z1↑.TEXT);
         Z1:= Z1↑.PREV;
      END;
  END; (* ANZEIGEN *)

BEGIN
  LEERELISTE;
  REPEAT
    ANZEIGEN;
    WRITELN('E infuegen');
    WRITELN('L oeschen');
    WRITELN('B eenden');
    WRITELN;
```

Bild 96: Eine doppelt verkettete Liste (Fortsetzung nächste Seite)

```
      WRITE('==> ');READLN(C);
      CASE C OF
       'E': EINFUEGEN;
       'L': LOESCHEN;
       ELSE (* nichts *)
      END; (* CASE *)
    UNTIL C='B';
  END.
```

Bild 96: Eine doppelt verkettete Liste (Schluß)

Verwaltung einer sortierten Kundenliste

In Bild 98 ist ein komplettes Programm angegeben, das die am Anfang des Abschnittes erwähnte Kundenliste implementiert. Im Unterschied zu den bisher vorgestellten Demonstrationsprogrammen mit Zeigervariablen wird die Kundenliste ständig alphabetisch sortiert gehalten.

Alle Funktionen sind so zu Modulen zusammengefaßt und ausführlich kommentiert, daß Sie die einzelnen Operationen mit den folgenden Hinweisen nachvollziehen können:

Suche

- Will man in einer Liste ein Element löschen oder einfügen, so muß man das Feld NAECHSTER beim **Vorgänger** korrigieren. Deshalb werden in der Suchroutine VORHANDEN zwei Zeiger verwendet. Dabei hinkt der Zeiger Z beim Durchlaufen der Liste immer ein Record hinter dem Zeiger Z1 her.

Leere Anfangs- und End-Records

- Normalerweise muß man das Einfügen des ersten Elementes in der Liste und das Löschen des letzten Elementes in einer Liste explizit programmieren, da hierbei andere Zeiger umgesetzt werden müssen als bei allen anderen Operationen. Um diese Sonderbehandlung zu vermeiden, wird im Programm die Liste um ein unbenutztes erstes und letztes Element erweitert. Diese Records sind in Bild 97 schraffiert dargestellt.

Marke am Listenende

- Dieses letzte Element wird auch zur Aufnahme einer Marke bei der Suchroutine VORHANDEN verwendet (siehe auch Abschnitt 2.9 über die Suche im Array).
- Die Ordnung wird beim Einfügen aufrechterhalten, indem die Einfügeposition durch Suchen in der Liste bestimmt wird. Zum Einfügen des Namens »Ross« in der (sortierten) Liste

  `Arnold Jackson Jones Overmars Vollmann`

Einfügen

  durchläuft man die Namen so lange, bis man den Namen erreicht, hinter dem »Ross« eingefügt werden soll. Zeigt die Variable VOR auf den Namen »Overmars« und NEU auf den neuen Record mit dem Namen »Ross«, so fügt man NEU folgendermaßen hinter VOR in die Liste ein: (siehe Bild 97.A)

```
NEU↑.NAECHSTER:= VOR↑.NAECHSTER;
VOR↑.NAECHSTER:= NEU;
```

Löschen

– Beim Löschen wird eine Variable zweimal dereferenziert:
  `VOR↑.NAECHSTER:= VOR↑.NAECHSTER↑.NAECHSTER`
  Dadurch wird die in Bild 97.B dargestellte Operation durchgeführt.

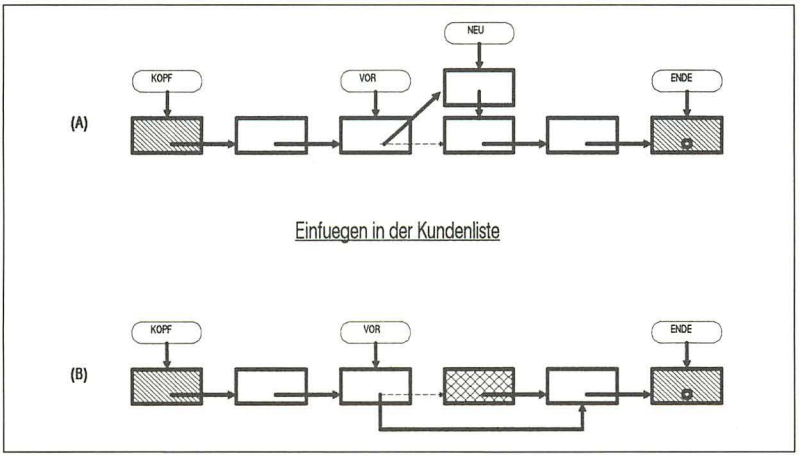


Bild 97:
A) Einfügen in der Kundenliste
B) Löschen in der Kundenliste

```
PROGRAM KUNDENLISTE (INPUT, OUTPUT);
(* BEISPIEL FUER DIE VERWALTUNG EINER LISTE MIT ZEIGERN.   *)
(* DIE DATEN WERDEN STAENDIG SORTIERT IN EINER LISTE GE-   *)
(* HALTEN. JEDER RECORD BESITZT DAZU EINEN ZEIGER AUF      *)
(* DEN ALPHABETISCHEN NACHFOLGER. UM DAS EINFUEGEN UND     *)
(* LOESCHEN EINFACH ZU GESTALTEN, BESITZT DIE LISTE JE     *)
(* EIN LEERES ELEMENT AM ANFANG UND ENDE.                  *)

  CONST LEN = 10;                  (* LAENGE EINES NAMENS     *)

  TYPE STRING = STRING[LEN];
       KUNDENZEIGER = ↑ KUNDE;
       KUNDE   = RECORD
                   NAME      : STRING;
                   KNUMMER   : INTEGER;
                   NAECHSTER: KUNDENZEIGER;
                 END;

  VAR KOPF: KUNDENZEIGER;          (* KOPF DER KUNDENLISTE    *)
      ENDE: KUNDENZEIGER;          (* ENDE DER KUNDENLISTE    *)
      CH  : CHAR;                  (* BENUTZEREINGABE         *)

  FUNCTION VORHANDEN(S:STRING; VAR Z:KUNDENZEIGER):BOOLEAN;
  (* SUCHT NAME (S) IN DER LISTE. ERGEBNIS=TRUE, FALLS       *)
  (* S GEFUNDEN WURDE. Z ZEIGT BEI RUECKKEHR IMMER AUF       *)
  (* DIE POSITION DES ALPHABETISCHEN VORGAENGERS.            *)
    VAR Z1: KUNDENZEIGER;        (* Z1 STEHT IMMER EIN RECORD*)
                                 (* WEITER ALS DER ZEIGER Z  *)
  BEGIN
    Z:= KOPF; Z1:= KOPF↑.NAECHSTER;
    ENDE↑.NAME:= S;              (* MARKE AM LISTENENDE      *)
    WHILE Z1↑.NAME<S DO
      BEGIN
        Z:= Z1; Z1:= Z1↑.NAECHSTER
      END;
    VORHANDEN:= (Z1↑.NAME=S) AND (Z1<>ENDE)
  END; (* VORHANDEN *)
```

Bild 98:
Das Programm Kundenliste
(Fortsetzung nächste Seite) ►

```
PROCEDURE DRUCKE(Z: KUNDENZEIGER);
(* DRUCKE DEN INHALT DES REFERENZIERTEN RECORDS          *)
BEGIN
 WITH Z↑ DO
   WRITELN("NAME:",NAME:LEN+2,"  NUMMER:",KNUMMER:5)
END; (* DRUCKE *)

PROCEDURE EINGABE;
(* EINGABE EINES NEUEN KUNDENRECORDS                     *)
  VAR N  : STRING;
      NEU: KUNDENZEIGER;     (* ZEIGER AUF NEUEN RECORD  *)
      VOR: KUNDENZEIGER;     (* ZEIGER AUF ALPABETISCHEN *)
                             (* VORGAENGER IN DER LISTE  *)
BEGIN
  WRITE("NAME:"); READLN(N);
  IF VORHANDEN(N,VOR) THEN
     WRITELN(N," IST BEREITS KUNDE!")
  ELSE
     BEGIN
      NEW(NEU);              (* NEUEN RECORD BESORGEN    *)
      WRITE("KUNDENNUMMER:");
      READLN(NEU↑.KNUMMER); (* UND BELEGEN              *)
      NEU↑.NAME:= N;
      NEU↑.NAECHSTER:= VOR↑.NAECHSTER;
      VOR↑.NAECHSTER:= NEU; (* NEU NACH VOR EINFUEGEN   *)
    END
END; (* EINGABE *)

PROCEDURE AUSGABE;
(* AUSGABE EINES KUNDENRECORDS                           *)
  VAR N  : STRING;
      VOR: KUNDENZEIGER;     (* ZEIGER AUF ALPABETISCHEN *)
                             (* VORGAENGER IN DER LISTE  *)
BEGIN
  WRITE("NAME:"); READLN(N);
  IF VORHANDEN(N,VOR) THEN
     DRUCKE(VOR↑.NAECHSTER)
  ELSE
     WRITELN(N,"NICHT ALS KUNDE GESPEICHERT!")
END; (* AUSGABE *)

PROCEDURE LOESCHEN;
  VAR N  : STRING;
      VOR: KUNDENZEIGER;     (* VORGAENGER IN DER LISTE  *)
BEGIN
  WRITE("NAME:"); READLN(N);
  IF VORHANDEN(N,VOR) THEN
     BEGIN
      WRITELN("GELOESCHT WURDE:");
      DRUCKE(VOR↑.NAECHSTER);
                             (* NACHFOLGER VON VOR AUS   *)
                             (* DER LISTE STREICHEN:     *)
      VOR↑.NAECHSTER:= VOR↑.NAECHSTER↑.NAECHSTER;
     END
  ELSE
     WRITELN(N,"NICHT ALS KUNDE GESPEICHERT!")
END; (* LOESCHEN *)

PROCEDURE TABELLE;
(* DRUCKE EINE ALPHABETISCHE LISTE ALLER KUNDEN          *)
  VAR Z:KUNDENZEIGER;
BEGIN
```

Bild 98: Das Programm Kundenliste

```
      Z:= KOPF↑.NAECHSTER;          (* Z AUF ANFANG DER LISTE    *)
      WHILE Z<>ENDE DO              (* SOLANGE NICHT LETZTEN     *)
        BEGIN                       (* (LEEREN) RECORD ERREICHT:*)
          DRUCKE(Z);
          Z:=Z↑.NAECHSTER           (* ZUM NAECHSTEN KUNDEN      *)
        END;
    END; (* TABELLE *)

  BEGIN (* HAUPTPROGRAMM *)
    NEW(KOPF); NEW(ENDE);           (* ANFANG UND ENDE BILDEN    *)
    KOPF↑.NAECHSTER:=ENDE;          (* LISTE IST LEER            *)
    REPEAT                          (* EINGABESCHLEIFE           *)
      WRITELN("E INGABE");
      WRITELN("A USGABE");
      WRITELN("L OESCHEN");
      WRITELN("T ABELLE");
      WRITELN("X BEENDEN");
      READLN(CH);
      CASE CH OF
        "T": TABELLE;
        "E": EINGABE;
        "A": AUSGABE;
        "L": LOESCHEN;
        "X": ;
       ELSE WRITELN("UNGUELTIGE WAHL")
      END;
    UNTIL CH="X";
  END.
```

Bild 98: Das Programm Kundenliste (Schluß)

Speicherplatzrückgewinnung mit Freispeicherlisten

Zum Abschluß des Abschnittes sollen Sie noch ein Standardverfahren kennenlernen, mit dem man den Speicherplatz, der durch das Löschen von Records frei wird, wiederverwenden kann. Die Idee besteht darin, die (logisch) gelöschten Records zu einer neuen Liste, der **Freispeicherliste**, zu verketten. Vor jedem Aufruf der Prozedur NEW prüft man dann, ob sich nicht bereits ein unbenutzter Record in der Freispeicherliste befindet.

Einfügungen und Löschungen in der Freispeicherliste erfolgen am einfachsten am selben Ende, so daß diese Liste also ein LIFO (*stack*) ist. Um diese Freispeicherverwaltung in das Programm Kundenliste zu integrieren, muß man folgende Änderungen vornehmen: Zunächst deklariert man einen Zeiger auf den Kopf der Freiliste:

```
VAR FREI: KUNDENZEIGER;
```

Anschließend werden die eigentlichen Prozeduren zur Verwaltung der Freispeicherliste definiert (siehe Operationen Löschen und Einfügen bei Kellerspeichern!):

```
PROCEDURE NEWKUNDE (VAR Z: KUNDENZEIGER);
(* Liefere Zeiger auf neuen Kundenrecord *)
BEGIN
  IF FREI = NIL THEN
     (*Freispeicherliste ist leer *)
     NEW(Z)
```

```
  ELSE
    BEGIN (* Entferne erstes Element aus der Freiliste *)
      Z:= FREI; FREI:= FREI↑.NAECHSTER;
    END;
END; (* NEWKUNDE *)

PROCEDURE DISPOSEKUNDE (Z: KUNDENZEIGER);
(* Speicherplatz von Z ist freigeworden. Z↑ darf nicht *)
(* mehr verwendet werden; Erweitere die Freiliste      *)
BEGIN
  Z↑.NAECHSTER:= FREI; FREI:= Z;
END; (* DISPOSEKUNDE *)
```

Jetzt müssen die Routinen nur korrekt aufgerufen werden. Dazu ersetzt man den Aufruf NEW(NEU) in der Prozedur EINGABE durch NEWKUNDE(NEU). Um in der Prozedur LOESCHEN den Nachfolger von VOR zu löschen, merkt man sich zunächst in einer Variablen ALT den Zeiger auf das zu löschende Objekt. Dann kann man den Record wie gehabt aus der Verzeigerung der Kundenliste entfernen und zum Schluß mit DISPOSEKUNDE(ALT) den Record ALT↑ in die Freispeicherliste einfügen.

```
PROCEDURE LOESCHEN;
VAR N  : STRING;
    VOR: KUNDENZEIGER; (*VORGAENGER IN DER LISTE *)
    ALT: KUNDENZEIGER;
BEGIN
  WRITE('NAME:'); READLN(N),
  IF VORHANDEN(N,VOR) THEN
    BEGIN
      WRITELN('GELOESCHT WURDE:');
      DRUCK(VOR↑.NAECHSTER);
      ALT:= VOR↑.NAECHSTER;
      VOR↑.NAECHSTER:= ALT↑.NAECHSTER;
      DISPOSEKUNDE(ALT);
    END
  ELSE
    WRITELN(N,' NICHT ALS KUNDE GESPEICHERT!');
END; (* LÖSCHEN *)
```

Natürlich muß die Freispeicherliste am Programmanfang korrekt initialisiert werden. Da sie zu diesem Zeitpunkt noch leer ist, erhält der Zeiger auf den Listenanfang der Wert NIL:

```
FREE:= NIL
```

Dieses Verfahren der Verwaltung von freigewordenem Speicher ist sehr effizient, so daß man meist die Verwendung systemspezifischer Speicherverwaltungsprozeduren (DISPOSE, MARK, RELEASE) vermeiden kann. Die Prozeduren MARK und RELEASE in *Pascal 2.0* sind in der Dokumentation in Kapitel 4.4.4.4 beschrieben.

2.18.2 Bäume

Bäume als nichtlineare Datenstrukturen

Zum Abschluß dieser Einführung in die Programmiersprache Pascal soll noch ein einziges Beispiel für eine nichtlineare dynamische Datenstruktur mit Zeigern gegeben werden: Ein **Baum** ist (in der Graphentheorie) ein Graph mit einem Eingang, in dem jeder Knoten auf genau einem Weg vom Eingang aus erreicht werden kann (siehe Bild 99).

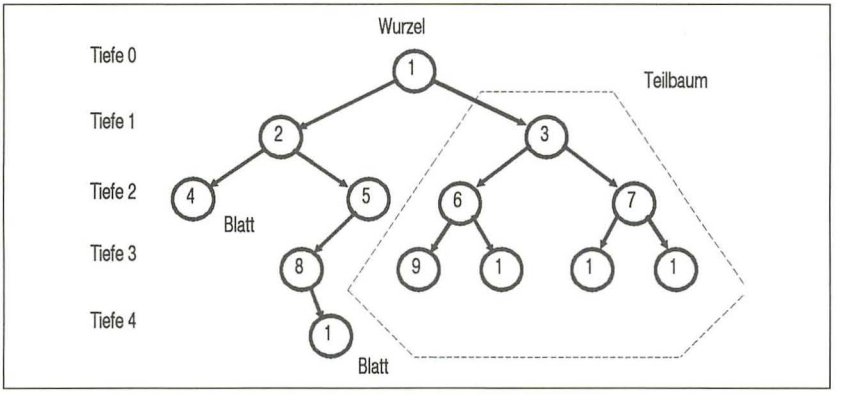


Bild 99: Ein Baum

Beschreibung der Elemente eines Baumes

Bäume zeichnet man üblicherweise mit der **Wurzel** (dem Eingang) nach oben. Jeder Knoten besitzt eine gewisse **Tiefe**, das ist die Distanz zum Eingang (hier also Knoten 1). Knoten ohne Nachfolger bezeichnet man als **Blätter**. Jeder innere Knoten hat eine gewisse Anzahl an direkten Nachfolgern, die selbst Wurzeln von **Teilbäumen** sind. So besitzt der Wurzelknoten (1) zwei direkte Nachfolger (2 und 3), wobei z.B. 3 die Wurzel des Teilbaumes aus den Knoten 3, 6, 7, 9, 10, 11 und 12 bildet. Die Maximalanzahl der direkten Nachfolger, die ein Knoten in einem Baum besitzt, heißt der Grad des Baumes.
Bild 99 zeigt einen **binären** Baum, also einen Baum des Grades 2, da jeder Knoten nur 0,1 und 2 direkte Nachfolger besitzt. Knoten mit mindestens einem Nachfolger, die also nicht Blätter sind, heißen innere Knoten. Außerdem läßt sich eine **Ordnung** auf den Knoten des Baumes definieren, indem man für jeden Knoten K im Baum folgende Relationen vereinbart:

Definition einer Ordnung

O1. Alle Knoten im linken Teilbaum mit der Wurzel K sind kleiner als K.
O2. Alle Knoten im rechten Teilbaum mit der Wurzel K sind größer als K.

Mit diesen Regeln ergibt sich in Bild 99 für die Wurzel folgende Relation:
2, 4, 5, 8, 13 < 1 < 3, 6, 7, 9, 10, 11, 12
Wendet man diese Regeln (rekursiv) auch auf alle Knoten in den Teilbäumen an, so erhält man eine vollständige Ordnung:
4 < 2 < 8 < 13 < 5 < 1 < 9 < 6 < 10 < 3 < 11 < 7 < 12
Nun sind alle Begriffe definiert, um zu erläutern, wie sich die Kundenverwaltung des letzten Abschnittes in einem *binären Baum* effizient organisieren läßt. Wiederum sollen alle Kunden alphabetisch sortiert gespeichert werden, um ohne Nachsortieren eine nach Namen sortierte Liste auszugeben. Vor allen Dingen wird die oben beschriebene Ordnung auch benutzt, um einen Kunden mit möglichst wenigen Vergleichen über seinen Namen zu finden.

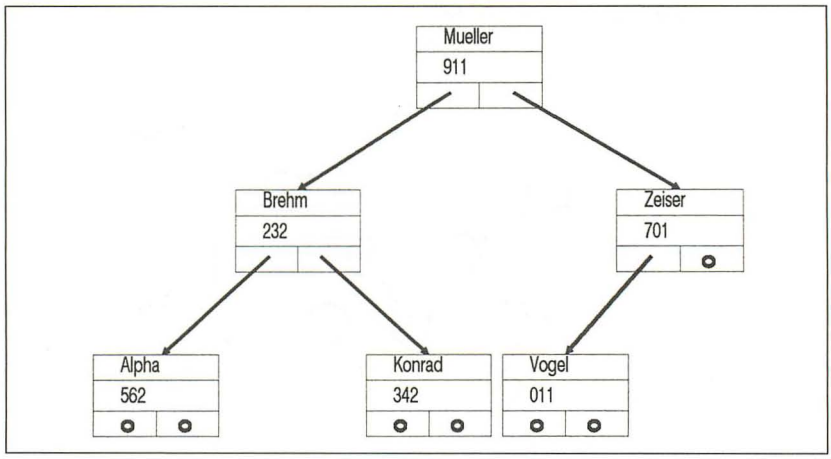


Bild 100: Baum mit Kundenrecords

Deklaration eines Knotentyps

Einfüge- und Löschoperationen sollen so erfolgen, daß eine Datenstruktur wie in Bild 100 entsteht. Diese Struktur läßt sich durch die folgende Typ- und Variablendeklaration in Pascal erzeugen:

```
TYPE KUNDENZEIGER = ↑ KUNDE;
     KUNDE = RECORD
                NAME      : STRING[10];
                KNUMMER   : INTEGER;
                L,R       : KUNDENZEIGER;
             END;
VAR  WURZEL : KUNDENZEIGER;
```

Die Knoten des Baumes werden also durch Records des Typs KUNDE dargestellt. Die Datenfelder NAME und KNUMMER sind wie bei der Kundenliste im letzten Abschnitt definiert. Jedoch erhält jeder Kunde nun **zwei** Nachfolger. Die Felder L und R enthalten deshalb Zeiger auf den linken und den rechten Nachfolger im Baum. Existiert einer der beiden Nachfolger nicht, so wird das entsprechende Feld mit dem Wert NIL belegt.

Die Operationen mit dieser Datenstruktur sind in Bild 101 ausführlich kommentiert wiedergegeben. Bevor Sie weiterlesen, sollten Sie das Programmlisting studieren, um sich mit dem Problem vertraut zu machen.

Initialisierung

Am Programmanfang sind noch keine Kunden gespeichert und der Baum ist leer:

```
WURZEL:= NIL
```

Einfügen

Beim Einfügen muß die alphabetische Reihenfolge im Baum erhalten bleiben. Hierzu durchläuft man den Baum von der Wurzel ausgehend zu den Blättern. Dabei wird in jeder Tiefe der Name des neuen Kunden mit dem im Knoten gespeicherten Namen verglichen. Ist der Name *größer*, so muß die Einfügung im rechten Teilbaum ausgeführt werden, sonst wird der Zeiger zum linken Teilbaum verfolgt.

Um »KUNZE« in den Baum von Bild 100 einzufügen, bestimmt man in der Prozedur EINFUEGEN den folgenden Weg:

```
KUNZE < MUELLER          gehe links
KUNZE > BREHM            gehe rechts
KUNZE > KONRAD           gehe rechts
```

Da kein rechter Nachfolger von »KONRAD« existiert, das heißt der Zeiger R im Record ist NIL, kann man jetzt »KUNZE« als rechten Nachfolger von »KONRAD« einfügen.

Rekursion zur Suche

Diese Strategie beschreibt die rekursive Prozedur EINFUEGEN. Sie wird mit zwei Parametern aufgerufen. NEU ist ein Record vom Typ KUNDE, um den der Baum erweitert werden soll. Der Variablenparameter Z ist ein Zeiger auf die Wurzel des Teilbaumes, in den NEU eingefügt werden soll. Bitte beachten Sie, daß hier ein Variablenparameter erforderlich ist, um bei einer Einfügung (Z = NIL) den Zeiger auf das neue Element in der aufrufenden Umgebung zu ändern.

Laufzeitvergleich zwischen Liste und Baum

Bei der Suche fällt auf, daß man im Idealfall die Größe des noch zu durchsuchenden Teilbaumes in jedem Schritt halbiert. Somit hat man in einem Baum mit N Knoten nach log(N) Schritten die Einfügeposition bestimmt. Natürlich muß man dafür sorgen, daß der entstehende Baum **ausgeglichen** bleibt, so daß jeder innere Knoten möglichst gleich viele linke wie rechte Nachfolger besitzt. Würde man zum Beispiel ständig Einfügungen am rechten Blatt vornehmen, so hätte der entstehende Baum die Form einer Liste, die über das Feld R verkettet ist. Dann beträgt die Tiefe des Baumes N statt log(N) im Falle des ausgeglichenen Baumes.

Ausgabe in der Sortierreihenfolge

Zur Ausgabe der alphabetisch geordneten Tabelle muß man den Baum in der durch die Regeln O1 und O2 definierten Ordnung durchlaufen. Dies geschieht am elegantesten mit der rekursiven Prozedur INORDER. Der Parameter Z gibt die Wurzel des Teilbaumes an, der gedruckt werden soll. Die oben beschriebene Ordnung verlangt eine Ausgabe in der folgenden Reihenfolge:

Rekursive Lösung der Ausgabe

1. Ausgabe aller Knoten im linken Teilbaum (Z↑.L)
2. Ausgabe des Wurzelknotens Z↑
3. Ausgabe aller Knoten im rechten Teilbaum (Z↑.R)

Da die Wurzel in der Ordnung zwischen dem linken und dem rechten Teilbaum liegt, heißt diese Ordnung *inorder* (im Gegensatz zu *postorder* und *preorder*).

Löschungen müssen die Ordnung erhalten

Löschungen eines inneren Knotens im geordneten binären Baum erfordern etwas genauere Überlegungen, damit die Ordnung zwischen den übrigbleibenden Knoten erhalten bleibt. Zunächst wird wie beim Einfügen das zu löschende Element in der Prozedur ENTFERNE rekursiv gesucht. Beim Löschen von Z↑ müssen nun folgende zwei Fälle unterschieden werden:

1. Z besitzt nur einen Nachfolger. In diesem Fall kann Z durch diesen linken oder rechten Nachfolger ersetzt werden.
2. Ansonsten muß Z durch den größten Wert Z1 in seinem linken Teilbaum ersetzt werden, da Z1 einerseits größer als alle anderen Elemente im linken Teilbaum und gleichzeitig kleiner als alle Elemente im rechten Teilbaum ist.

Bei der Operation (2) wird die rekursive Prozedur HOLEHOCH verwendet. Den größten Wert in einem Teilbaum findet man offensichtlich, indem man bei jedem inneren Knoten nach rechts geht. Damit ist Z1 entweder ein Blatt oder ein Knoten, der nur einen linken Nachfolger besitzt, so daß sich Z1 selbst leicht löschen läßt.

Sollten Sie Interesse an solchen Datenstrukturen und Algorithmen gefunden haben, können Sie sich, ausgerüstet mit den Kenntnissen aus diesem Buch, unbesorgt der Fachliteratur über Systematische Programmierung [2], [5], [6], [7], [8] (siehe Anhang E) zuwenden. Dort finden Sie viele weitere Anregungen und vollständige Algorithmen, die effizient in Pascal realisiert werden können.

```
PROGRAM KUNDENBAUM (INPUT, OUTPUT);
(* VERWALTUNG DER KUNDENDATEN IN EINEM NACH NAMEN          *)
(* GEORDNETEN BINAERBAUM.                                   *)

  CONST LEN = 10;                  (* LAENGE EINES NAMENS    *)

  TYPE STRING = STRING[LEN];
       KUNDENZEIGER = ↑ KUNDE;
       KUNDE   = RECORD
                   NAME     : STRING;
                   KNUMMER  : INTEGER;
                   L, R     : KUNDENZEIGER;
                 END;

  VAR WURZEL: KUNDENZEIGER;        (* WURZEL DES BAUMES      *)
      CH    : CHAR;                (* FUNKTIONS-AUSWAHL      *)

  PROCEDURE DRUCKE(Z: KUNDENZEIGER);
  (* DRUCKE DEN INHALT DES REFERENZIERTEN RECORDS           *)
  BEGIN
   WITH Z↑ DO
     WRITELN("NAME:",NAME:LEN+2,"  NUMMER:",KNUMMER:5)
  END; (* DRUCKE *)
```

Bild 101: Kundenverwaltung in einem Baum

```
PROCEDURE EINGABE;
(* EINGABE EINES NEUEN KUNDENRECORDS                          *)
  VAR K: KUNDE;

  PROCEDURE EINFUEGEN(NEU:KUNDE; VAR Z:KUNDENZEIGER);
  (* FUEGE NEU AN DER KORREKTEN POSITION IM TEILBAUM MIT*)
  (* DER WURZEL Z EIN.                                        *)
  BEGIN
    IF Z=NIL THEN              (* TEILBAUM IST LEER:         *)
      BEGIN                    (* FUEGE NEU ALS BLATT EIN    *)
        NEW(Z);Z↑:= NEU;       (* BELEGE Z↑ MIT NAME  UND    *)
        Z↑.L:= NIL;            (* KUNDENNUMMER. Z↑ HAT       *)
        Z↑.R:= NIL             (* KEINE NACHFOLGER !         *)
      END
    ELSE
      BEGIN                    (* VERGLEICH DER SCHLUESSEL:*)
        IF NEU.NAME=Z↑.NAME THEN
          WRITELN(NEU.NAME," IST BEREITS KUNDE!")
        ELSE
          IF NEU.NAME<Z↑.NAME THEN
            EINFUEGEN(NEU,Z↑.L) (* IM LINKEN  ODER          *)
          ELSE
            EINFUEGEN(NEU,Z↑.R) (* IM RECHTEN TEILBAUM     *)
      END                        (* EINFUEGEN               *)
  END; (* EINFUEGEN *)

BEGIN  (* EINGABE *)
  WRITE("NAME:"); READLN(K.NAME);
  WRITE("KUNDENNUMMER:"); READLN(K.KNUMMER);
  EINFUEGEN(K, WURZEL);        (* K IM BAUM EINFUEGEN        *)
END; (* EINGABE *)

PROCEDURE INORDER(Z: KUNDENZEIGER);
(* DRUCKE IN ALPHABETISCHER REIHENFOLGE DIE KNOTEN DES  *)
(* TEILBAUMES MIT WURZEL Z.                                 *)
BEGIN
  IF Z<>NIL THEN               (* TEILBAUM IST NICHT LEER: *)
    BEGIN
      INORDER(Z↑.L);           (* DRUCKE LINKEN TEILBAUM   *)
      DRUCKE (Z);              (* DIE WURZEL SELBST UND    *)
      INORDER(Z↑.R)            (* DANN DEN RECHTEN TEILBAUM*)
    END
END; (* TABELLE *)

PROCEDURE LOESCHE;
  VAR NAME: STRING;

  PROCEDURE ENTFERNE(N: STRING; VAR Z: KUNDENZEIGER);
  (* ENTFERNE DEN KUNDEN MIT NAME N AUS DEM TEILBAUM     *)
  (* MIT DER WURZEL Z                                     *)

    PROCEDURE HOLEHOCH(VAR Z1: KUNDENZEIGER);
    (* ERSETZE Z DURCH DEN GROESSTEN WERT IM LINKEN      *)
    (* TEILBAUM Z↑L                                       *)
    BEGIN
      IF Z1↑.R=NIL THEN
        BEGIN                  (* KOPIERE FELDER NACH Z↑  *)
          Z↑.NAME    := Z1↑.NAME;
          Z↑.KNUMMER:= Z1↑.KNUMMER;
          Z1:=Z1↑.L            (* ERSETZE Z1 DURCH SEINEN *)
                               (* LINKEN NACHFOLGER       *)
        END
      ELSE                     (* RECHTS WEITERSUCHEN:    *)
        HOLEHOCH(Z1↑.R)
    END; (* HOLEHOCH *)
```

Bild 101: Kundenverwaltung in einem Baum (Fortsetzung nächste Seite)

```
      BEGIN  (* ENTFERNE *)
        IF Z=NIL THEN
          WRITELN(N," IST NICHT GESPEICHERT!")
        ELSE
          IF N=Z↑.NAME THEN       (* ERSETZE Z DURCH EINEN     *)
            BEGIN                 (* SEINER NACHFOLGER         *)
              IF Z↑.L= NIL THEN Z:=Z↑.R ELSE
              IF Z↑.R= NIL THEN Z:=Z↑.L
              ELSE  HOLEHOCH(Z↑.L)
            END
          ELSE                    (* SUCHE IN DEN TEILBAEUMEN *)
            IF N<Z↑.NAME THEN ENTFERNE(N,Z↑.L)
                          ELSE ENTFERNE(N,Z↑.R)
      END;(* ENTFERNE *)

    BEGIN (* LOESCHE *)
      WRITE("NAME:"); READLN(NAME);
      ENTFERNE(NAME,WURZEL)       (* LOESCHE KUNDEN IM BAUM    *)
    END; (* LOESCHE *)

  BEGIN (* HAUPTPROGRAMM *)
    WURZEL:= NIL;                 (* BAUM IST LEER             *)
    REPEAT                        (* EINGABESCHLEIFE           *)
      WRITELN("T ABELLE");
      WRITELN("E RWEITERN");
      WRITELN("L OESCHEN");
      WRITELN("X BEENDEN");
      READLN(CH);
      CASE CH OF
        "T": INORDER(WURZEL);     (* DRUCKE GESAMTEN BAUM      *)
        "E": EINGABE;
        "L": LOESCHE;
        "X": ;
       ELSE WRITELN("UNGUELTIGE WAHL")
      END;
    UNTIL CH="X";
  END.
```

Bild 101: Kundenverwaltung in einem Baum (Schluß)

Aufgaben

1. Versuchen Sie, eine Kundenliste ohne leere Records am Anfang und Ende der Liste zu verwalten. Sollte die Programmlänge dabei um mehr als dreißig Zeilen anwachsen, haben Sie zumindest den Sinn dieser Hilfsrecords erkannt.
2. Alternativ können Sie die Kundenliste alphabetisch sortiert und doppelt verkettet anlegen. Dann können Sie auch die Prozedur VORHANDEN so ändern, daß sie einen Zeiger auf das alphabetisch größere Element liefert, da Sie immer einen Zeiger auf den Vorgänger besitzen. Eventuell können Sie auch auf die leeren Records am Anfang und Ende der Liste verzichten.

3. Um zu prüfen, ob Sie die Operationen in Bild 101 im großen und ganzen verstanden haben, sollten Sie das Programm so ändern, daß die Records nach *Kundennummer* sortiert im Baum gespeichert werden.
4. Schreiben Sie eine rekursive Prozedur REVERSE, die eine Liste invertiert, so daß der letzte Record als erster in der neuen Liste erscheint. (Diese rekursive Lösung ist nur für kurze Listen geeignet!)
5. Schreiben Sie eine rekursive Prozedur SWAP, die in einem binären Baum den linken und rechten Nachfolger jedes inneren Knotens vertauscht. Bild 102 zeigt das Ergebnis dieser Operation für den Baum aus Bild 99:

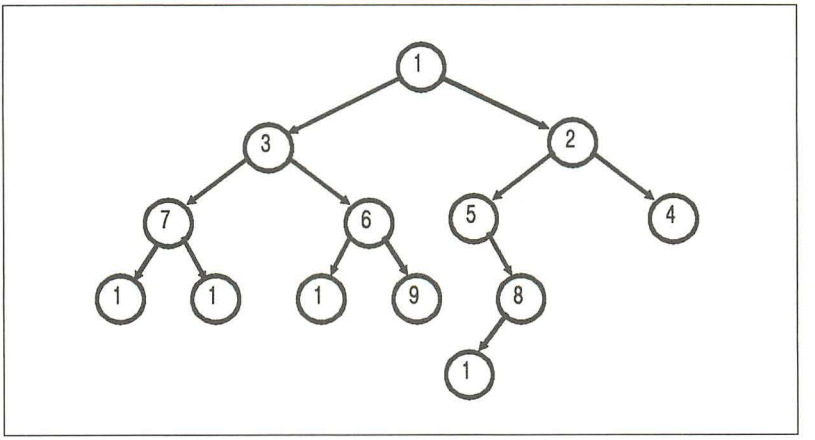


Bild 102: Wirkung der Prozedur SWAP

6. Schreiben Sie eine Prozedur MERGE, die zwei sortierte Listen A und B zu einer sortierten Liste C zusammenfügt (siehe Programm MERGE zum Sortieren von Files!).
   `PROCEDURE MERGE(A,B: KUNDENZEIGER; VAR C: KUNDENZEIGER);`
   Beispiel:
   A = (Alpha, Brehm, Konrad, Mueller, Vogel, Zeiser)
   B = (Bauer, Friedrich, Schmidt, Schulze)
   C = (Alpha, Bauer, Brehm, Friedrich, Konrad, Mueller, Schmidt, Schulze, Vogel, Zeiser)
7. Ersetzen Sie die Prozedur INORDER durch die Prozedur PREORDER, die die Knoten eines Baumes in der folgenden Reihenfolge druckt:
   - Drucke die Wurzel
   - Drucke alle Blätter des linken Teilbaumes in PREORDER
   - Drucke alle Blätter des rechten Teilbaumes in PREORDER

```
PROCEDURE PREORDER(Z: KUNDENZEIGER);
BEGIN
  IF Z<>NIL THEN
    BEGIN
      DRUCKE(Z);
      PREORDER(Z↑.L);
      PREORDER(Z↑.R);
    END;
END; (* PREORDER *)
```

Verdeutlichen Sie sich die Reihenfolge der Ausgabe an einigen Beispielen! Anschließend wird es Ihnen sicher leichtfallen, die Prozedur POSTORDER in Pascal zu definieren, die den Wurzelknoten bei der Ausgabe zuletzt nennt.

Kapitel 3
Tips und Tricks

Im Gegensatz zu Kapitel 2 stehen in diesem Kapitel die Besonderheiten des C128 im Vordergrund. Dabei sollen nicht alle POKE-, PEEK- und SYS-Befehle, die seit Jahren verschiedene Zeitschriften zum C64 und C128 füllen, einfach in Pascal formuliert werden, sondern nur exemplarisch der Zugriff auf Routinen in Maschinensprache gezeigt werden. Im ersten Abschnitt wird außerdem noch ausführlich auf die Verwaltung von Files eingegangen, da das Betriebssystem des C128 Eigenschaften besitzt, die über die rein sequentielle Datenspeicherung hinausgehen.

3.1 Files im C128-Modus

Vergleich BASIC–Pascal

Haben Sie bereits in BASIC mit Dateien gearbeitet, so werden Sie bei der Verwendung der OPEN- und CLOSE-Routinen von *Pascal 2.0* sicher auf keinerlei Probleme stoßen. Sollten Sie jedoch erst durch Abschnitt 2.16 Interesse an Files gewonnen haben, so sind sicherlich einige Hinweise zur Verwaltung von Dateien beim C128 angebracht. Aber auch der fortgeschrittene Programmierer findet am Ende dieses Abschnittes einige interessante Hinweise zu Relativdateien, die nicht durch den im *report* definierten Sprachumfang erfaßt werden.

Verwaltung von Files beim C128

Zunächst muß die grundsätzliche Arbeitsteilung beim *file handling* auf Commodore-Computern erläutert werden. Bei Heimcomputern dieser Firma (VC20, C64, C128) befindet sich im ROM neben dem BASIC-Interpreter ein Betriebssystemkern, der die Kommunikation des Computers mit allen Peripheriegeräten (Bildschirm, Tastatur, Floppy, Drucker...) organisiert. Als *normaler Anwender* haben Sie über die Sprache BASIC nur einen indirekten Zugang zu den Betriebssystemroutinen. Ein übersetztes Pascal-Programm verwendet zur Ausführung von File-Operationen ebenfalls Teile des im ROM gespeicherten Betriebssystems. Jedoch ist das Betriebssystem des C128 im folgenden Sinne *erweiterbar*: Viele der anschließbaren Peripheriegeräte sind (wie z.B. die Floppy) *intelligent*. Das heißt, sie besitzen eigene Programme, um die vom Computer kommenden Daten und Befehle selbständig zu verarbeiten.

Einheitliche Ein- und Ausgabe über logische Dateien

Um so unterschiedliche Geräte wie die Tastatur, eine Floppy oder einen Plotter einheitlich behandeln zu können, bietet das Betriebssystem das Konzept der **logischen Dateien**: Nur beim Öffnen eines Files müssen Sie angeben, welches physikalische Medium Sie wie ansprechen möchten. Anschließend können Sie auf alle geöffneten Dateien einheitlich lesend und schreibend zugreifen. Die Datenein- und -ausgabe beenden Sie mit einem CLOSE-Befehl.

File-Variable = logische Datei

Eine logische Datei wird in *Pascal 2.0* durch eine File-Variable repräsentiert. Mit den Prozeduren PUT, WRITE und WRITELN können Sie an Ausgabe-Files Daten senden. Diese Datenausgabe erfolgt zeichenweise. Ebenfalls Byte für Byte erfolgt die Dateneingabe mit GET, READ und READLN. Welche Struktur die übertragenen Daten haben (z.B. Texte oder Files mit Recordkomponenten) wird in Pascal durch die Deklaration der File-Variablen festgelegt. Jedoch muß diese Struktur auch korrekt von dem Peripheriegerät akzeptiert werden. Daher ist es sehr wichtig, daß Sie bei den OPEN-Befehlen in Pascal-Programmen korrekte Parameter wählen, um die Ein- und Ausgabegeräte richtig zu adressieren. Diese Parameter werden in diesem Abschnitt insbesondere für die Diskettenstation (Floppy) in Beispielen besprochen.

Parameter des OPEN-Befehls

Sie können bis zu 10 Files gleichzeitig bearbeiten. Für jede Datei muß ein eigener OPEN-Befehl durchgeführt werden, der sozusagen die File-Variable beim Betriebssystem *anmeldet*.

Die **Geräteadresse** ist eine ganze Zahl zwischen 0 und 15, die das Peripheriegerät bestimmt, auf dem die logische Datei physikalisch gespeichert werden soll:

0 Tastatur (nur als Eingabegerät)
1 Kassettenrecorder (Ein- und Ausgabegerät)
2 RS232-Schnittstelle (Ein- und Ausgabegerät)
3 Bildschirm (nur als Ausgabegerät)
4 Drucker oder Plotter an der seriellen Schnittstelle
8 Floppy an der seriellen Schnittstelle
>8 weitere Geräte an der seriellen Schnittstelle

Sekundäradressen der Floppy

Viele Geräte können unter der gleichen Geräteadresse mehrere **Sekundäradressen** besitzen. Diese Adresse im Bereich zwischen 0 und 255 ist stark von dem verwendeten Gerätetyp abhängig. In diesem Kapitel soll nur die Diskettenstation besprochen werden. Dort gelten folgende Konventionen über die Sekundäradresse:

0, 1 sind reservierte Sekundäradressen, die zum Laden und Speichern von Programmen mit LOAD und SAVE verwendet werden.

15 ist die Sekundäradresse des sogenannten Kommandokanals der Floppy

2 bis 14 sind Sekundäradressen, die Sie in Pascal-Programmen zur Speicherung sequentieller Dateien verwenden können.

Haben Sie bei einer Floppy gleichzeitig mehrere Files geöffnet, so müssen diese **verschiedene** Sekundäradressen besitzen, damit das Betriebssystem die Dateien bei Schreib- und Leseoperationen voneinander unterscheiden kann. Schließlich müssen Sie bei einem OPEN-Befehl für Diskettendateien immer einen **File-Namen** (max. 16 Zeichen lang) nennen. Durch ein Suffix legen Sie dabei gleichzeitig den Modus fest, in dem Sie die Datei bearbeiten möchten. Besitzen Sie ein Doppellaufwerk, so können Sie im File-Namen auch die **Laufwerknummer** (»0:« oder »1:«) für das File spezifizieren.

Beispiele:

```
VAR FILE1,FILE2: TEXT;
    FILE3      : FILE OF INTEGER;
    FILE4      : FILE OF RECORD;
                          A: INTEGER;
                          B: STRING[30];
                        END;
OPEN(FILE1, 8, 3, '1:DATEN,SEQ,READ');
OPEN(FILE2, 8, 4, '0:TEST,SEQ,WRITE');
OPEN(FILE3, 8, 5, '0:INTEGER,USR,APPEND');
OPEN(FILE4, 8, 6, '@0:RECORDS,PRG,WRITE');
```

Überschreiben existierender Files

Die Laufwerkbezeichnung »1:« ist natürlich nur dann sinnvoll, wenn Sie ein Doppellaufwerk besitzen. Stellen Sie dem File-Namen das Zeichen CHR(64) (»@«) voran, so wird am Ende der Ausgabe auf das File eine eventuell zuvor existierende Version des Files gelöscht. Jedoch wird der Platz, den die alte Version auf der Diskette belegte, nicht freigegeben. Deshalb kann beim Schreiben nicht die gesamte Speicherkapazität der Diskette genutzt werden. Man löscht daher besser die alte Version des Files vor dem Eröffnen einer neuen Datei (siehe Abschnit 3.3, Routine SCRATCH).

Hinter der Laufwerkbezeichnung folgt der eigentliche File-Name, unter dem die Datei im Inhaltsverzeichnis der Diskette aufgeführt wird. Er darf keine Sonderzeichen und kein Komma enthalten.

Filetypen der Floppy

Nach einem Komma wird der Typ der Datei spezifiziert. *Normale* Pascal-Dateien, die nur sequentiell lesend und schreibend bearbeitet werden, erhalten sinnvollerweise die Typen SEQ (*sequential*) oder USR (*user defined*). Den Typ PRG (*program file*) können Sie bei Bedarf ebenfalls verwenden, jedoch können solche Files im Inhaltsverzeichnis nicht mehr eindeutig identifiziert werden. Der Typ REL (*relative*) wird später in diesem Kapitel separat behandelt.

Lesen, Schreiben, Erweitern

Getrennt durch ein weiteres Komma folgt schließlich der Modus, in dem das File bearbeitet werden soll. Sollen von einer Datei Daten (mit GET, READ, READLN) gelesen werden, so muß der Modus READ gewählt werden. Zum Schreiben einer neuen Datei wird der Modus WRITE gewählt. Somit lassen sich die Operationen RESET und REWRITE des *report* mit OPEN-Befehlen

ersetzen. Eine Besonderheit stellt der Modus APPEND dar. Bei Ausgabeoperationen auf ein in diesem Modus eröffnetes File werden die Daten an ein bereits auf der Diskette existierendes File dieses Namens angehängt.
Wie bereits erwähnt, sollten Sie beim Lesen einer Datei darauf achten, daß die Deklaration der File-Variablen, mit denen die Daten geschrieben und gelesen werden, übereinstimmen. Da alle Daten für das Betriebssystem aus unstrukturierten Byte-Folgen bestehen, existiert keine Möglichkeit, auch innerhalb von Files Typüberprüfungen durchzuführen.

Speicherung der Kundenliste (Bild 98) auf Diskette

Mit den folgenden Prozeduren können Sie das Programm KUNDENLISTE erweitern, so daß die Daten beim Programmende auf Diskette gespeichert werden, um bei einem nachfolgenden Programmstart gelesen zu werden.

```
CONST FILENAME = '0:KLIST';
TYPE KUNDENDATEI = FILE OF KUNDE;

PROCEDURE WRITEFILE;
(* Diese Prozedur wird am Ende des Hauptprogrammes aufge- *)
(* rufen, um den Inhalt der Kundenliste alphabetisch sor- *)
(* tiert auf Diskette zu speichern.                        *)
  VAR KFILE: KUNDENDATEI;
      K    : KUNDENZEIGER;
  BEGIN
  OPEN(KFILE, 8, 3, '@' + FILENAME + ',SEQ,WRITE');
  K:= KOPF↑.NAECHSTER; (* leeren Anfangsrecord ignorieren *)
  WHILE K<>ENDE DO   (* bis zum leeren Enderecord:      *)
    BEGIN
      KFILE↑:= K↑; PUT(KFILE);
    END;
  CLOSE(KFILE);
END; (* WRITEFILE *)

PROCEDURE READFILE;
(* Diese Prozedur wird anstelle der Zuweisung              *)
(* KOPF↑.NAECHSTER:= ENDE aufgerufen, um die Kundenliste   *)
(* von Diskette zu lesen.                                  *)
  VAR KFILE: KUNDENDATEI;
  K,K1 : KUNDENZEIGER;
BEGIN
  OPEN(KFILE, 8, 3, FILENAME + ',SEQ,READ');
  GET(KFILE); K1:= KOPF;
```

Bild 103: Erweiterung des Programmes KUNDENLISTE

```
  WHILE NOT EOF(KFILE) DO
    BEGIN
      NEW(K); K↑:= KFILE↑;
      K1↑.NAECHSTER:= K; K1:= K;
      GET(KFILE);
    END;
  CLOSE(KFILE); K1↑.NAECHSTER:= ENDE;
END; (* READFILE *)
```

Bild 103: Erweiterung des Programmes KUNDENLISTE (Schluß)

Abfrage des Floppy-Status

Bei Ein- und Ausgaben auf Files wie in Bild 103 sollte man zumindest nach dem Aufruf der Prozeduren OPEN und CLOSE prüfen, ob Verarbeitungsfehler bei der Floppy aufgetreten sind (Diskette war nicht eingelegt oder schreibgeschützt). Dazu können Sie die Routine DSTATUS aus Abschnitt 3.3 verwenden, die den Kommandokanal der Floppy abfragt.

Sortierung des Inhaltsverzeichnisses einer Diskette

Ein etwas spezielleres Beispiel (Bild 104) geht auf die besonderen Eigenschaften der Floppy ein. In BASIC ist es möglich, das Directory der Diskette mit dem Befehl

LOAD"$0",8

im Format eines BASIC-Programmes zu laden. Mit der Prozedur READALL werden die Namen und Typen aller Files auf der Diskette gelesen und in einer Tabelle (T) gespeichert. Durch wiederholte Aufrufe dieser Prozedur kann der Inhalt mehrerer Disketten gesammelt werden. Anschließend werden die File-Namen mit der Prozedur QUICK alphabetisch sortiert. Zur Druckerausgabe der sortierten Tabelle T wird die Prozedur AUSGABE verwendet.

```
PROGRAM DISKSORT(INPUT,OUTPUT);
(* FORMATIERTER AUSDRUCK DES DISKINHALTES        *)
(* 12.10.1986 FLORIAN MATTHES    PASCAL 2.0       *)

CONST MAXP=250;    (* LAENGE DER NAMENSTABELLE   *)

TYPE  FILETYP=(PRG,SEQ,USR,REL);
      EINTRAG=RECORD
                NAME: STRING[16];
                TYP : FILETYP;
                BLK : INTEGER;
                ID  : ARRAY[0..1] OF CHAR;
               END;
VAR   P      : INTEGER;
      WEITER : BOOLEAN;
      T      : ARRAY[0..MAXP] OF EINTRAG;

 PROCEDURE READALL;
 (* DISKETTEN-DIRECTORY KOMPLETT EINLESEN.        *)
 (* INFORMATIONEN IN T AB INDEX P ABLEGEN. P      *)
 (* WIRD GLOBAL VERAENDERT!                       *)
  VAR A,ID1,ID2: CHAR;
```

Bild 104: Directory lesen (Fortsetzung nächste Seite) ►

```
      I,N        : INTEGER;
      INF        : TEXT;
      ENDE       : BOOLEAN;

  FUNCTION BYTE:INTEGER;
  (* ZEICHEN LESEN UND IN BYTE WANDELN          *)
    VAR C:CHAR;
  BEGIN
    READ(INF,C);
    IF EOLN(INF) THEN BYTE:=13
                 ELSE BYTE:=ORD(C)
  END; (* BYTE *)

  FUNCTION NUMBER:INTEGER;
  (* L UND H-BYTE LESEN UND UMWANDELN           *)
  BEGIN
    NUMBER:= BYTE + 256*BYTE
  END; (* NUMBER *)

BEGIN (* READALL *)
 OPEN(INF,8,0,'$0');    (* INHALTSVERZEICHNIS    *)
                        (* VON LAUFWERK 0        *)
 WRITELN; WRITELN;
 FOR I:=1 TO 32 DO      (* TITELZEILE AUSWERTEN:*)
  BEGIN READ(INF,A);
    IF I IN [9..24] THEN WRITE(A);
    IF I=27 THEN ID1:=A;
    IF I=28 THEN ID2:=A
  END;
 WRITELN; ENDE:= FALSE;
 WHILE (P<=MAXP) AND NOT ENDE DO
 BEGIN
  N:=NUMBER; N:= NUMBER;    (* N=ANZAHL BLOECKE *)
  (* ANFUEHRUNGSZEICHEN ODER'BLOCKS FREE'LESEN *)
  REPEAT
    ENDE:= EOF(INF)
  UNTIL (BYTE=34) OR ENDE;
  IF NOT ENDE THEN          (* ALLES EINTRAGEN:  *)
   WITH T[P] DO
    BEGIN NAME:=''; READ(INF,A);
     WHILE A<>'"' DO
       BEGIN NAME:= NAME + A; READ(INF,A) END;
     REPEAT
       READ(INF,A);         (* EINTRAG TYP:      *)
     UNTIL A IN ['P','S','U','R'];
     CASE A OF
       'P':TYP:=PRG;
       'S':TYP:=SEQ;
       'U':TYP:=USR;
       'R':TYP:=REL
     END; (* CASE *)
     REPEAT UNTIL BYTE=0;     (* BIS ZEILENENDE *)
     BLK:=N;
     ID[0]:= ID1; ID[1]:= ID2;
     P:= P+1;
    END
 END;
 CLOSE(INF);
END; (* READALL *)

PROCEDURE QUICK(L,R:INTEGER);
(* TABELLE NACH NAMEN AUFSTEIGEND SORTIEREN    *)
 VAR I,J:INTEGER;
     X,W:EINTRAG;
```

Bild 104: Directory lesen

```
BEGIN
 IF L<R THEN
  BEGIN
   I:=L; J:=R; X:=T[(L+R)DIV 2];
   REPEAT
    WHILE T[I].NAME<X.NAME DO I:=I+1;
    WHILE X.NAME<T[J].NAME DO J:=J-1;
    IF I<=J THEN
    BEGIN
     W:=T[I]; T[I]:=T[J]; T[J]:=W;
     I:=I+1; J:=J-1
    END;
   UNTIL I>J;
   QUICK(L,J); QUICK(I,R)
  END;
END; (* QUICK *)

FUNCTION OK:BOOLEAN;
  VAR C:CHAR;
BEGIN
 WRITELN('  (JA ODER NEIN)'); WRITE('==>');
 REPEAT READLN(C) UNTIL C IN ['J','N'];
 OK:= C='J'
END; (* OK *)

PROCEDURE AUSGABE;
 VAR J, ZPROSEITE: INTEGER;
     PRT          : TEXT;
BEGIN
 WRITELN; WRITELN;
 WRITELN('DRUCKER BEREIT?');
 IF OK THEN
  BEGIN
   OPEN(PRT,4,0);
   WRITE('ZEILEN PRO SEITE: ');READLN(ZPROSEITE);
   FOR J:=0 TO P-1 DO
    WITH T[J] DO
     BEGIN
      IF J MOD (ZPROSEITE-2)=0 THEN
       BEGIN
        WRITE(PRT,'| NAME:              | TYP ');
        WRITELN(PRT,'| BLK  | ID |');
        WRITE(PRT,'| ─────              | ─── ');
        WRITELN(PRT,'| ─── | ── |');
       END;
       WRITE(PRT,'|',NAME:17,' |');
       CASE TYP OF
        PRG:WRITE(PRT,'PRG':4);
        SEQ:WRITE(PRT,'SEQ':4);
        USR:WRITE(PRT,'USR':4);
        REL:WRITE(PRT,'REL':4)
       END;
       WRITELN(PRT,' |',BLK:4,' |',ID:3,' |');
      END;
     CLOSE(PRT)
    END
 END;(* AUSGABE *)

BEGIN(* MAIN *)
 P:=0;                              (* TABELLE LEER *)
 WRITELN(CHR(147),'DISK-SORT':24);
 WRITELN('──────────':24);
 REPEAT
  WRITELN; WRITELN;
  WRITELN('WEITERE DISKETTEN?'); WEITER:=OK;
```

Bild 104: Directory lesen (Fortsetzung nächste Seite)

```
IF WEITER THEN READALL;
 UNTIL NOT WEITER;
 WRITELN; WRITELN;
 WRITELN ('*** BITTE WARTEN ***');
 QUICK(0,P-1);
 AUSGABE;
END. (* MAIN *)
```

Bild 104: Directory lesen (Schluß)

Einlesen eines Bytes mit READ

Besonders beachten sollten Sie die Funktion BYTE: Da das Inhaltsverzeichnis von einer Datei mit dem Typ TEXT

```
VAR INF: TEXT;
```

gelesen wird, wandelt das Pascal-Laufzeitsystem alle Zeilenendezeichen (CHR(13), *carriage return*) in Leerzeichen (CHR(20)) um. In diesem Fall ist EOLN(INF)=TRUE. Um nun beliebige Bytes aus dem Bereich 0..255 zu lesen, wird in der Funktion BYTE eine Umwandlung des gelesenen Zeichens in eine INTEGER-Zahl vorgenommen.

Verwaltung einer Relativdatei auf Diskette

Die Floppy besitzt ein eigenes Betriebssystem, das die Verwaltung der Daten in Blöcken auf der Floppy vornimmt und unter anderem auch das Inhaltsverzeichnis selbständig verwaltet. Dieses Betriebssystem kennt einen weiteren File-Typ, der bei OPEN-Befehlen gewählt werden kann. Es handelt sich hierbei um **Relativdateien**. Während die übrigen Dateien (Typ PRG, SEQ und USR) nur sequentiell lesend oder schreibend bearbeitet werden können, sind auf Relativdateien auch indizierte Zugriffe (wie bei einem Array) möglich.

Deklaration

Voraussetzung für solche *indexsequentiellen Dateien* ist, daß jedes Element des Files die gleiche Länge in Bytes besitzt. Dies ist jedoch in der Sprache Pascal automatisch durch die Deklaration

```
VAR DATEI = FILE OF Komponententyp
```

Indizierter Zugriff über die Recordnummer

gesichert. Sinnvollerweise verwendet man Relativdateien für Files, bei denen der Komponententyp ein zusammengesetzter Typ (z.B. Record) ist. Die Komponenten besitzen innerhalb des Files eine eindeutige *Recordnummer*. Nachdem eine Relativdatei angelegt wurde, können die Komponenten **in beliebiger Reihenfolge** durch die Angabe ihrer Recordnummer indiziert geschrieben und gelesen werden. Die erste Komponente im File besitzt die Recordnummer 1. Andererseits können Relativdateien auch sequentiell Komponente für Komponente in der Reihenfolge der Recordnummern (1,2,3 ...) bearbeitet werden. Dies wird erreicht, indem der Zeiger auf die aktuell bearbeitete Komponente bei jeder Lese- und Schreiboperation auf die Folgekomponente im File gesetzt wird.

```
PROGRAM BOOKS (INPUT, OUTPUT);
(* DEMONSTRATIONSPROGRAMM ZUR VERWALTUNG EINER RELATIV-    *)
(* DATEI IN PASCAL 2.0.                                     *)

  CONST DEVICE = 8; (* GERAETEADRESSE DER FLOPPY            *)
        SECOND = 3; (* SEKUNDAERADRESSE DER RELATIVDATEI    *)
        NAME   = '0:BUCHTITEL';
        DEFAULTSIZE = 50;

  TYPE  BUCH = RECORD
                 TITEL   : STRING[30];
                 AUTOR   : STRING[30];
                 SIGNATUR: INTEGER;
                 MARKE   : CHAR;            (* SIEHE TEXT! *)
               END;

  VAR REL   : FILE OF BUCH;  (* RELATIVE DATEI              *)
      DSK   : TEXT;          (* KOMMANDO-KANAL DER FLOPPY *)
      WAHL  : CHAR;

 PROCEDURE DSTATUS;
 (* FEHLERKANAL DER FLOPPY AUSLESEN, FEHLER IM KLARTEXT    *)
 (* MIT FEHLERNUMMER, SPUR UND SEKTOR ANZEIGEN:            *)
   VAR NUMMER: INTEGER;
       TEXT  : STRING[40];
 BEGIN
  READ(DSK, NUMMER);
  IF NUMMER>=20 THEN
    BEGIN
      READLN(DSK, TEXT); WRITELN(NUMMER:3, TEXT);
    END;
 END; (* DSTATUS *)

 PROCEDURE SEEK (INDEX: INTEGER);
 (* SETZE DEN FILEZEIGER IM FILE 'REL' AUF DIE KOMPONENTE *)
 (* MIT DER RECORDNUMMER 'INDEX'  (>=1).                  *)
 BEGIN
   WRITE(DSK,'P',              (*  BEFEHL: POINTER SETZEN  *)
    CHR(SECOND),               (*  IN DIESER DATEI         *)
    CHR(INDEX AND 255),        (*  AUF DIESEN RECORD       *)
    CHR(HBYTE(INDEX)),         (*  (LOW UND HIGH-BYTE)     *)
    CHR(1));                   (*  AUF DAS ERSTE BYTE IN   *)
                               (*  DER KOMPONENTE TRECORD  *)
    DSTATUS;
 END;(* SEEK *)

 PROCEDURE INITREL;
 (* ANLEGEN EINER NEUEN RELATIVDATEI. EINE EVENTUELL      *)
 (* BEREITS EXISTIERENDE DATEI WIRD GELOESCHT.            *)
   VAR DNAME : STRING[20];
       I     : INTEGER;
 BEGIN
   WRITELN(DSK,'S' + NAME);        (* LOESCHE ALTE DATEI   *)
   (* NEUE DATEI MIT SATZLAENGE ANLEGEN:                   *)
   DNAME:= NAME + ',L,' + CHR(SIZEOF(BUCH));
   OPEN(REL, DEVICE, SECOND, DNAME);
   DSTATUS;
   WRITELN('DATEI WIRD INITIALISIERT');
   WITH REL↑ DO
     BEGIN
       TITEL   :='';
       AUTOR   :='';
       SIGNATUR:=0;
       MARKE   := CHR(13);                      (* S. TEXT *)
     END;
   SEEK(1);                              (* ZUM DATEIANFANG *)
```

Bild 105: Relativdateien in Pascal 2.0 (Fortsetzung nächste Seite) ►

```
    FOR I:= 1 TO DEFAULTSIZE DO
      PUT(REL);
    CLOSE(REL); DSTATUS;
  END; (* INITREL *)

  PROCEDURE READRECORD;
  BEGIN
    GET(REL); DSTATUS;
    WITH REL↑ DO
      BEGIN
        WRITELN('TITEL:'    :10, TITEL);
        WRITELN('AUTOR:'    :10, AUTOR);
        WRITELN('SIGNATUR:':10, SIGNATUR);
      END
  END; (* READRECORD *)

  PROCEDURE WRITERECORD;
  BEGIN
    WITH REL↑ DO
      BEGIN
        WRITE('TITEL:'    :10); READLN(TITEL);
        WRITE('AUTOR:'    :10); READLN(AUTOR);
        WRITE('SIGNATUR:':10); READLN(SIGNATUR);
        MARKE:= CHR(13);
      END;
    PUT(REL); DSTATUS
  END; (* WRITERECORD *)

  PROCEDURE POSITION;
  (* POSITIONIERE DEN ZEIGER IM FILE AUF DEN DURCH DEN      *)
  (* BENUTZER GEWAEHLTEN RECORD:                            *)
    VAR NUMMER: INTEGER;
  BEGIN
    READLN(NUMMER);  SEEK(NUMMER);
  END; (* POSITION *)

BEGIN
  OPEN(DSK, DEVICE, 15, 'I0');    (* KOMMANDOKANAL FLOPPY *)

  WRITELN(#147, 'BUCHDATEI:':25);
  WRITELN(       '==========':25);
  WRITELN;
  WRITELN('MOECHTEN SIE EINE NEUE DATEI ANLEGEN?');
  WRITELN;
  WRITE('==>N'  #157); READLN(WAHL);
  IF WAHL<>'N' THEN INITREL;

  OPEN(REL, DEVICE, SECOND, NAME + ',REL'); DSTATUS;
  SEEK(1);                                (* ZUM DATEIANFANG *)
  WRITELN;
  WRITELN;
  WRITELN('JETZT KOENNEN DIE  SAETZE DURCH');
  WRITELN('EINGABE DER SATZNUMMER (#)');
  WRITELN('BEARBEITET WERDEN:');
  REPEAT
    WRITELN;
    WRITELN('WAEHLEN SIE:');
    WRITELN('X    BEENDEN');
    WRITELN('L # LESEN   ');
    WRITELN('S # SCHREIBEN');
    WRITE('==>'); READ(WAHL);
    CASE WAHL OF
      'X':;
      'L': BEGIN POSITION; READRECORD   END;
      'S': BEGIN POSITION; WRITERECORD  END
      ELSE WRITELN('  UNGUELTIGE WAHL!')
```

Bild 105: Relativdateien in Pascal 2.0

```
      END;(* CASE *)
    UNTIL WAHL='X';
    CLOSE(REL); DSTATUS;
    CLOSE(DSK)
  END.
```

Bild 105: Relativdateien in Pascal 2.0 (Schluß)

Verwaltung von Buchtiteln in einer Relativdatei

Das Programm in Bild 105 zeigt ein vollständiges Programm zur Verwaltung von Buchtiteln in einer Relativdatei. Natürlich ist dieses Programm noch weit davon entfernt, eine sinnvolle Anwendung für Relativdateien zu zeigen, vielmehr sollen nur die generellen Operationen mit Relativdateien anschaulich illustriert werden.

Jedes Buch wird durch einen Record des Typs BUCH in der Relativdatei REL beschrieben. Jeder Record wird innerhalb des Files durch eine ganzzahlige Recordnummer größer als Null identifiziert. Der Benutzer des Programmes muß also für die Bücher eindeutige Recordnummern vergeben. Der Typ BUCH speichert folgende Felder

Der Datentyp BUCH

Titel	30 Zeichen
Autor	30 Zeichen
Signatur	ganze Zahl (dies ist **nicht** die Recordnummer!)
MARKE	1 Zeichen

Marke am Satzende

Das Feld MARKE hat eine rein technische Bedeutung: Das Betriebssystem der Floppy speichert normalerweise nur Texte (in Pascal wären dies ARRAYS OF CHAR) in einer relativen Datei. Diese Texte werden grundsätzlich mit dem Zeichen *carriage return* (CHR(13)) beendet. Fehlt dieses Zeichen am Ende eines Records, so ignoriert das Betriebssystem unter Umständen die letzten binären Nullen innerhalb des Records. Daher wird vor allen Schreiboperationen auf das File REL das Feld MARKE mit dem Wert CHR(13) belegt.

Neben der eigentlichen Relativdatei

```
REL: FILE OF BUCH;
```

existiert der sogenannte **Kommandokanal** der Floppy (Sekundäradresse 15):

```
DSK: TEXT;
```

Über ihn werden Kommandos vom Computer an die Floppy in Form von Zeichenfolgen gesendet. Außerdem kann man von diesem Kommandokanal auch Informationen über die Ausführung von File-Operationen erhalten (siehe Prozedur DSTATUS). Im Programm BOOKS werden folgende Befehle verwendet:

Diskettenbefehle über den Kommandokanal

```
'IO'
'S' + NAME
'P' + CHR(SECOND) + CHR(INDEX AND 255) + CHR(HBYTE(INDEX))
+ CHR(1)
```

Der erste Befehl (*initialize*) initialisiert die Diskette im Laufwerk 0 der Floppy.

Dadurch werden eventuell noch geöffnete Floppy-Kanäle geschlossen. Der zweite Befehl (*scratch*) löscht die Datei mit dem File-Namen NAME. Besonders wichtig für die Arbeit mit Relativdateien ist der dritte Befehl. Mit ihm ist es möglich, in dem File mit der Sekundäradresse SECOND den Schreib- und Lesezeiger auf das Element mit der Recordnummer INDEX zu setzen. Während der gesamten Programmausführung bleibt die Datei DSK geöffnet. Grundsätzlich sollte sie als letztes File in einem Programm geschlossen werden.

Anlegen einer Relativdatei mit festen Satzlängen

Beim ersten Aufruf des Programms existiert noch keine Relativdatei. Sie wird erst in der Prozedur INITREL explizit angelegt. Zunächst wird der Dateiname für den OPEN-Befehl gebildet:

```
DNAME:= NAME + ',L' + CHR(SIZEOF;(BUCH));
```

Bei dem nachfolgenden OPEN-Befehl

```
OPEN(REL, DEVICE, SECOND, DNAME)
```

wird deshalb eine relative Datei mit einer **Satzlänge** erzeugt, die exakt der Größe eines Records vom Typ BUCH in Bytes entspricht. DEVICE ist wie üblich die Geräteadresse der Floppy (8). Die Sekundäradresse (SECOND = 3) ermöglicht es, über den Kommandokanal beim Befehl »P« gezielt diese Relativdatei zu spezifizieren.

Bei einer Relativdatei wird normalerweise erst dann Speicherplatz auf der Diskette für ein Record des Files reserviert, falls versucht wird, hinter dem File-Ende zu schreiben. Durch die Prozeduraufrufe

```
SEEK(50); PUT(REL)
```

würde der 50. Record der Datei REL gespeichert werden. Da dieser Record noch nicht existiert, würden auf der Floppy die Records mit den Indizes 1, 2, ... 49 mit zufälligen Werten belegt werden. Da diese Operation recht zeitaufwendig ist und eventuell nicht mehr genügend Speicherplatz auf der Diskette vorhanden sein könnte, wird statt dessen in der Prozedur INITREL folgende Schleife verwendet:

```
SEEK(1); FOR:= 1 TO DEFAULTSIZE DO PUT(REL);
```

Vorbelegung der Relativdateien

Da zuvor der Record REL↑ mit definierten Daten vorbelegt wurde, sind damit die ersten 50 Records korrekt initialisiert. Erst wenn durch spätere Schreiboperationen die File-Größe über DEFAULTSIZE Bücher anwächst, wird auf der Floppy zusätzlicher Speicherplatz reserviert.

Um eine bereits existierende Relativdatei zu benutzen, ist es nicht erforderlich, die Satzlänge erneut anzugeben, statt dessen wählt man beim OPEN-Befehl den Typ REL

```
OPEN(REL, DEVICE, SECOND, NAME + ',REL')
```

Positionierung in der Relativdatei

In vielen Implementationen der Sprache Pascal existiert der Befehl SEEK, der einen wahlfreien Zugriff (*random access*) auf die Elemente eines Files ermöglicht. In Anlehnung an diesen Standard wurde der Name der Prozedur SEEK

gewählt. Da die Verwaltung von Relativdateien ausschließlich Aufgabe des Betriebssystems der Floppy ist, genügt es, einen Positionierbefehl über den Kommandokanal der Floppy zu senden:

```
'P' + CHR(SECOND) + CHR(INDEX AND 255) + CHR(HBYTE(INDEX)) + CHR(1)
```

Dieser Befehl beginnt mit dem Zeichen »P«, dem die Sekundäradresse der Relativadresse (codiert als ein Zeichen) folgt. Anschließend wird die gewünschte Position in Form von zwei Byte übergeben. Zusätzlich besteht die Möglichkeit, innerhalb eines Records den Lesezeiger auf ein einzelnes Byte zu setzen. In der Sprache Pascal gibt es hierfür keine sinnvolle Anwendung, so daß die Position 1 (CHR(1)) gewählt wird. Nach dem Befehl »P« kann über den Kommandokanal das Ergebnis der Operation abgefragt werden:

```
00, OK, 00, 00
50, RECORD NOT PRESENT, 00, 00
```

Schreiben hinter dem Dateiende

Bewegen Sie den File-Zeiger hinter den momentan letzten Satz des Files, so erhalten Sie die Warnmeldung 50. Lesezugriffe (GET(REL)) auf nicht existente Records führen zu undefinierten Werten der Puffervariable REL↑, während bei Schreibzugriffen das File bis zur gewählten Komponente erweitert wird (was einige Sekunden dauern kann).

In der Prozedur READRECORD wird jeder Satz zweimal gelesen:

```
SEEK(P); GET(REL);
SEEK(P); GET(REL);
```

Der Grund hierfür liegt in einem Fehler des Betriebssystems der Commodore-Floppy: Bei Wechseln zwischen Lese- und Schreiboperationen auf Relativdateien, können unter Umständen Teile eines Satzes abgeschnitten werden. Bei einer Wiederholung des Lesevorganges tritt dieser Fehler nicht wieder auf.

Es ist extrem wichtig, daß eine eröffnete Relativdatei am Ende der Bearbeitung korrekt mit CLOSE geschlossen wird, damit alle Daten auch physikalisch auf Diskette gespeichert werden und nicht in Pufferspeichern verlorengehen.

Leider existieren einige Fehler im Betriebssystem **aller** Floppys für Commodore-Heimcomputer. Daher kann es teilweise zu Schwierigkeiten beim Erweitern und sequentiellen Lesen von Relativdateien kommen (Näheres findet man in Computerzeitschriften). Halten Sie sich jedoch an das Verarbeitungsschema von Bild 105, so können Sie Ihre Daten sicher in Relativdateien speichern.

Relativdateien und dynamische Datenstrukturen

Besonders interessant ist die Möglichkeit der Sprache Pascal, im Speicher des Computers dynamische Datenstrukturen (z.B. ausgeglichene Bäume) zu bilden, die eine schnelle Suche nach Schlüsseln (z.B. TITEL) ermöglichen. Mit jedem Schlüssel muß dann nur noch die Recordnummer gespeichert werden, unter der in einer relativen Datei auf Diskette die eigentliche Informationen zu diesem Schlüssel gespeichert sind. Somit läßt sich die hohe Speicherkapazität der Floppy mit der Geschwindigkeit der Sprache Pascal koppeln.

3.2 Pascal und Maschinensprache auf dem C128

Maschinennahe Programmierung erfordert Spezialwissen

Möchten Sie Pascal-Programme (z.B. aus Geschwindigkeitsgründen) um Unterprogramme in Maschinensprache erweitern, so benötigen Sie (neben einem Assembler) genauere Kenntnisse über die Speicherverteilung bei übersetzten Pascal-Programmen und die Darstellung von Variablen im Hauptspeicher. Diese Themen werden jeweils in den Abschnitten 4.3.4 und 4.4.2 behandelt.

Maschinennahe Operationen in Pascal 2.0

Neben der Möglichkeit, Registerinhalte beim Aufruf von Maschinencode-Programmen zu übergeben (siehe Beschreibung des Befehls SYS in Kapitel 4.4.4.8), können Sie Variablen auch an absolute Adressen in Bank 1 und der *common area* binden (siehe Abschnitt 4.4.2 »absolut adressierte Variablen«). Schließlich können Sie mit der Funktion ADR die absolute Adresse einer Variablen auf dem *stack* und dem *heap* bestimmen.

Durch Programmparameter (siehe Abschnitt 4.3.4) können Sie praktisch überall im Speicher Platz für Maschinenroutinen reservieren, die Sie beim Programmstart von Diskette nachladen können. Besonders benutzerfreundlich ist die folgende Methode, bei der der Maschinencode zusammen mit dem Pascalcode in Bank 0 gespeichert wird, so daß er mit den BASIC-Befehlen LOAD und SAVE automatisch mitgeladen wird.

Als Beispiel soll das folgende kleine Assembler-Programm aufgerufen werden:

Einfügen eines Maschinenprogramms zwischen Laufzeitsystem und Pascal-Code

```
$1CE9     JSR $FF7D        (Aufruf der Routine PRIMM)
          DFD 'HALLO WELT' , 0
          RTS              (Rückkehr zu PASCAL)
```

Es beginnt 10 Byte hinter dem Ende des Startcodes eines Pascal-Programmes ($1CE9 = 7391 + 10) und endet bei der Adresse $1CF8 = 7391 + 10 + 15. Den obigen Code können Sie direkt mit dem Monitor eingeben oder von einem assemblierten File absolut laden.

Dieses Mini-Assemblerprogramm soll von dem folgenden Pascal-Programm aufgerufen werden:

```
PROGRAM MCODE (OUTPUT; 8192);
  CONST PRINT = 7401; (* Startadresse M-Code *)
  VAR I: INTEGER;
BEGIN
  BANK:= 15; (* Code liegt in BANK 0 und benutzt Kernal *)
  FOR I:= 1 TO 5 DO
    SYS(PRINT);
END.
```

Verschieben des Codes eines Pascal-Programms

Die Zahl 8192 im Programmkopf veranlaßt den Compiler, den erzeugten Code ab der Adresse 8192 zu speichern. Dadurch wird verhindert, daß der Maschinencode (ab der Adresse 7401) überschrieben wird. Am Ende der Kompilation erhalten Sie folgende Meldung:

```
   P-CODE FROM 8192 TO 8238
CONSTANTS FROM 8192 TO 8192
```

Das heißt, der Programmcode belegt den Speicherbereich zwischen 8192 und 8238 in Bank 0. Gleichzeitig hat der Compiler den Zeiger auf das Ende der BASIC-Programme auf die Adresse 8238 gesetzt.

Modifikation des Laufzeitsystems

Nun muß dem Startcode für Pascal-Programme noch die geänderte Anfangsadresse des P-Codes mitgeteilt werden (siehe Abschnitt 4.3.4). Dies geschieht ebenfalls am einfachsten mit dem ROM-Monitor:

```
T +8192,+8200,+7391
```

Damit werden 8 Byte ab der Adresse 7391 (dem *normalen* P-Code Start) gespeichert, um das Pascal-Laufzeitsystem von der Änderung der Speicherverteilung zu unterrichten.

Damit sind Pascal-Programm und Maschinencode zu einer Einheit verknüpft, die nun auf Diskette gespeichert werden kann:

```
SAVE"BOTH",8
```

Ausführung der beiden Teile

Damit wird der Speicherbereich von 7169 bis 8238 gespeichert. Das Programm kann nun wie jedes Pascal-Programm von Diskette geladen und gestartet werden:

```
LOAD"BOTH",8
RUN
```

Absolute Adressen des Laufzeitsystems

Nachdem Sie nun wissen, wie Sie Maschinenprogramme zusammen mit Pascal-Programmen speichern können, sollen die folgenden Beispiele die Parameterübergabe zwischen den Routinen erläutern.

```
FUNCTION FREEMEM  : INTEGER;
  VAR TOPOFSTACK  : INTEGER [47];
      BOTTOMOFHEAP: INTEGER [53];
BEGIN
  FREEMEM:= ADDU (BOTTOMOFHEAP, - TOPOFSTACK);
END; (* FREEMEM *)
```

Die beiden Variablen TOPOFSTACK und BOTTOMOFHEAP werden vom Pascal-Laufzeitsystem verwaltet und markieren das untere und obere Ende des freien Speicherbereiches in Bank 1 zwischen *stack* und *heap*. Die Größe dieses Speicherbereiches (in Bytes) erhält man also einfach durch die Subtraktion der Adressen. Dabei wird die Funktion ADDU verwendet, um Bereichsüberschreitungen des Typs INTEGER zu ignorieren.

Absolute Variablen zur Kommunikation mit Maschinenprogrammen

Die Kommunikation mit absoluten Variablen erfolgt somit nach dem folgenden Schema: Im Pascal-Programm deklariert man eine Variable eines geeigneten Typs, die an eine absolute Adresse gebunden wird. Im Maschinencode kann man die dort gespeicherten Daten lesen und modifizieren:

```
PUFFER   EQU $FE
START    INC PUFFER
         BNE NOINC
         INC PUFFER+1
NOINC    RTS
```

Diese Routine erhöht also bei jedem Aufruf den Inhalt der 2 Byte-Variablen PUFFER an der Adresse $FE/$FF um 1. Zum Aufruf können Sie folgendes Programm verwenden:

```
PROGRAM CALLIT (OUTPUT);
  CONST START = ....; (* Anfangsadresse M-Code *)
  VAR PUFFER[254];    (* = $FE                 *)
BEGIN
  READLN(PUFFER);
  SYS(START)
  WRITELN(PUFFER);
  PUFFER:= 2 * PUFFER;
  SYS(START);
  WRITELN(PUFFER);
END.
```

Modifikation des heap

Absolut adressierte Variablen können ohne Einschränkungen in Funktionen, Prozeduren und Ausdrücken verwendet werden. Das folgende Programm modifiziert den Heappointer, um auf dem *heap* einen Speicherbereich für N-Bytes zu reservieren:

```
PROCEDURE GETMEM(N: INTEGER);
  VAR BOTTOMOFHEAP: INTEGER [53];
BEGIN
  IF FREEMEM<N THEN
    BEGIN
      WRITELN('OUT OF MEMORY ERROR'); HALT;
    END
  ELSE
    BOTTOMOFHEAP:= ADDU(BOTTOMOFHEAP, -N);
END; (* GETMEM *)
```

Insbesondere beim Aufruf von ROM-Routinen erfolgt die Parameterübergabe an Unterprogramme in Maschinensprache über Werte in den Prozessorregistern. Daher können Sie in *Pascal 2.0* den SYS-Befehl um eine Registervariable erweitern.

```
START     INX
          DEY
          LDA #0
          SEC
          RTS
```

Änderung der Prozessorregister in Pascal 2.0

Dieses Unterprogramm erhöht das X-Register um 1, dekrementiert das Y-Register, löscht den Akkumulator und setzt das Statusregister. Diese Änderungen der Prozessorregister lassen sich mit dem folgenden Pascal-Programm nachvollziehen:

```
PROGRAM REGISTER (INPUT,OUTPUT);
  CONST START = ...;  (* Startadresse der Routine *)
  VAR REGS: RECORD
              SR   : BYTE;   (* STATUSREGISTER *)
              AKKU: BYTE;   (* AKKUMULATOR    *)
              X    : BYTE;   (* X-REGISTER     *)
              Y    : BYTE;   (* Y-REGISTER     *)
            END;
BEGIN
  WITH REGS DO
    BEGIN
      WRITE('AKKU :'); READLN(AKKU);
      WRITE('X    :'); READLN(X);
      WRITE('Y    :'); READLN(Y);
    END;
  SYS(START,REGS); (* Register werden verändert *)
  WITH REGS DO
    BEGIN
      WRITE('AKKU :', AKKU:4);
      WRITE('X    :', X   :4);
      WRITE('Y    :', Y   :4);
      IF ODD(SR) THEN WRITE('Carry ');
      IF (SR AND 2)<>0 THEN WRITE ('Zero ');
      IF (SR AND 4)<>0 THEN WRITE ('IRQ ');
      ...
      IF (SR AND 128)<>0 THEN WRITE ('Negative ');
    END;
END.
```

Bild 106: Parameterübergabe mit Registervariablen

PEEK und POKE in Pascal 2.0

Natürlich existiert auch in *Pascal 2.0* die Möglichkeit, Parameter explizit mit POKE- und PEEK-Befehlen zu bearbeiten. Das folgende Programm verwendet z.B. die im C128 interruptgesteuert arbeitende 48-Bit-Uhr:

```
PROGRAM TIME (INPUT,OUTPUT);
  CONST TI = 160; (* 1/60 Sekunden Echtzeituhr *)
BEGIN
  BANK:= 15;
  POKE(TI,0); POKE (TI+1,0); POKE(TI+2,0);
  REPEAT
    WRITELN(PEEK(TI):4,PEEK(TI+1):4, PEEK(TI+2):4)
  UNTIL PEEK(TI+1) = 5;
END.
```

Zugriff auf den Bildschirmspeicher

Natürlich ist der Bildschirm auch in *Pascal 2.0* nicht vor hüpfenden Bällen sicher, die jedoch (trotz REAL-Arithmetik!) deutlich schneller als in BASIC fliegen. Vielleicht ist es recht eindrucksvoll, wenn Sie zum Vergleich die Variablen XK,YK sowie XSPEED und YSPEED als INTEGER-Variablen deklarieren (die Funktion INT wird dann natürlich nicht mehr bei der Berechnung von POS benötigt).

```
PROGRAM BOUNCE (OUTPUT);
  CONST B = 40;            (* Bildschirmbreite und -Höhe *)

        H = 25;
        BLANK = 32;     (* Bildschirmcode für Leerzeichen *)

        BALL  = 81;
        SCREENBASE = 1024;      (* Basisadresse Video RAM *)

  VAR XK,YK          : REAL;
      XSPEED, YSPEED: REAL;
      POS,LASTPOS    : INTEGER;
BEGIN
  SCNCLR; DISPLAY(1,10,10,'                    ',1);
  BANK:= 0;
  XSPEED:= RANDOM(0)*4;
  YSPEED:= 0.75;
  XK     := B DIV 2;                    (* Bildmitte *)
  YK     := H DIV 2;
  LASTPOS:= SCREENBASE;
```

Bild 107: Adressierung des Bildschirmspeichers

```
  REPEAT
   XK:= XK + XSPEED;
   YK:= YK + YSPEED;
   IF XK>=B THEN
     BEGIN XK:= 2*B-XK; XSPEED:= -XSPEED; END ELSE
   IF XK<0  THEN
     BEGIN XK:=-XK    ; XSPEED:= -XSPEED; END;
   IF YK>=H THEN
     BEGIN YK:= 2*H-YK; YSPEED:= -YSPEED; END ELSE
   IF YK<0  THEN
     BEGIN YK:=-YK    ; YSPEED:= -YSPEED END;
   POS:= SCREENBASE + INT(XK) + B * INT (YK);
   IF POS<>LASTPOS THEN
    BEGIN
      POKE(LASTPOS,BLANK);
      IF PEEK(POS)<>BLANK THEN
        BEGIN
          XSPEED:=-XSPEED; YSPEED:= -YSPEED;
        END;
      POKE(POS,BALL);
      LASTPOS:= POS;
    END;
  UNTIL KEYPRESSED;
  POKE(LASTPOS, BLANK);
 END.
```

Bild 107: Adressierung des Bildschirmspeichers (Schluß)

3.3 Nützliche Pascal-Routinen

Die folgende Liste zeigt eine Umsetzung einiger Befehle des BASIC 7.0 in Pascal-Prozeduren und -Funktionen. Dabei wurden die Sprite-Prozeduren nicht aufgenommen, da die dazu erforderliche interruptgesteuerte Unterbrechung von Pascal-Programmen bei Sprite-Kollisionen nicht realisiert werden kann.

Floppystatus prüfen

Um bei Dateioperationen den Floppystatus zu lesen, muß zuvor der Kommandokanal eröffnet worden sein:

```
VAR DSK: TEXT;
OPEN(DSK, 8, 15);
```

Anschließend kann der Fehlerkanal mit der Prozedur DSTATUS gelesen werden. Durch die Abfrage NUMMER > =20 wird die Meldung »00, OK, 00, 00« nicht als Fehler angezeigt:

```
PROCEDURE DSTATUS;
  VAR NUMMER: INTEGER;
      TEXT  : STRING[40];
BEGIN
  READ(DSK, NUMMER);
  IF NUMMER>=20 THEN
    BEGIN
      READLN(DSK, TEXT); WRITELN(NUMMER:3,' ',TEXT);
    END;
END; (* DSTATUS *)
```

APPEND

Diese Prozedur wird durch den Prozeduraufruf OPEN mit dem Schreibmodus APPEND realisiert (siehe Kapitel 3.1).

```
OPEN(F, 8, 3, 'TEST,SEQ,APPEND');
```

BACKUP

```
PROCEDURE BACKUP(SRC,DEST: CHAR; DEVICE: BYTE);
(* Kopiere Disketten auf einem Doppellaufwerk *)
  VAR DSK: TEXT;
BEGIN
  OPEN(DSK, DEVICE, 15, 'D' + DEST + '=D' + SRC);
  CLOSE(DSK)
END; (* BACKUP *)
```

Beispiel:

```
BACKUP('0','1',8)
```

COLLECT

```
PROCEDURE COLLECT(DRV: CHAR; DEVICE: BYTE);
(* Löscht unvollständige Dateien. Gibt Speicher frei *)
  VAR DSK: TEXT;
BEGIN
  OPEN(DSK, DEVICE, 15, 'V' + DRV);
  CLOSE(DSK)
END; (* COLLECT *)
```

Beispiel:

```
COLLECT('0',8)
```

CONCAT

```
PROCEDURE CONCAT(A,B,C:STRING; DEVICE:BYTE);
(* Verknüpfe die Dateien A und B in dieser Reihenfolge *)
(* zu einer neuen Datei C                              *)
  VAR DSK: TEXT;
BEGIN
  OPEN(DSK, DEVICE, 15, 'C' + C ' =' + A + ',' + B);
  CLOSE(DSK)
END; (* CONCAT *)
```

Beispiel:

```
CONCAT('0:TEXT1', '0:TEXT2', '0:TEXTNEU',8)
```

COPY

```
PROCEDURE COPY(A,B:STRING; DEVICE:BYTE);
(* Speichere eine Kopie der Datei A in einer neuen Datei B *)
  VAR DSK: TEXT;
BEGIN
  OPEN(DSK, DEVICE, 15, 'C' + B + ' =' + A);
  CLOSE(DSK)
END; (* COPY *)
```

Beispiel:

```
COPY('0:TEXTALT', '1:TEXTNEU', 8)
COPY('0:TEXT*',   '1:*',8);
```

Die Laufwerknummer 1 darf nur bei Doppellaufwerken gewählt werden.

DEC

```
FUNCTION DEC(S:STRING): INTEGER;
(* Umwandlung des hexadezimalen Strings S in eine Zahl *)
(* S darf nicht größer als $7FFF werden.               *)
  VAR SUM: INTEGER;
BEGIN
  SUM:= 0;
  WHILE LENGTH(S)>0 DO
    BEGIN
      SUM:= SUM * 16 + ORD(S[1]) - ORD('0');
      DELETE(S,1,1);
    END;
  DEC:= SUM;
END; (* DEC *)
```

DIRECTORY

Die Anzeige des Disk-Inhaltes existiert bereits als Routine in Maschinensprache. Zum Aufruf wird der SYS-Befehl verwendet, wobei im X-Register die logische File-Nummer der eröffneten Datei DIR übergeben wird.

```
PROCEDURE DIRECTORY(S:STRING; DEVICE: BYTE);
(* Disketteninhaltsverzeichnis mit Suchmuster S anzeigen *)
  VAR REGS: RECORD SR,A,X,Y:BYTE; END;
      DIR : TEXT;
BEGIN
  OPEN(DIR,DEVICE,0,S); BANK:=15;
  REGS.X:= HBYTE(STATUS(DIR));    (* logische Filenummer *)
  SYS(-24385,REGS);
END; (* DIRECTORY *)
```

Beispiele:

```
DIRECTORY('$',8);
DIRECTORY('$0:TEST*',8);
DIRECTORY('$*=REL',9);
```

Möchten Sie den Inhalt des Directory jedoch in Variablen speichern, so können Sie sich an dem Programm in Bild 104 orientieren.

HEADER

```
PROCEDURE HEADER(S:STRING; DEVICE: BYTE);
(* Lösche oder formatiere Diskette in Gerät mit der Nummer *)
(* DEVICE. S definiert den neuen Diskettennamen. Enthält   *)
(* dieser nach einem Komma zwei Zeichen, so wird eine      *)
(* Diskette mit dieser ID formatiert. Ansonsten wird nur   *)
(* das Inhaltsverzeichnis der Diskette gelöscht            *)
  VAR DSK: TEXT;
BEGIN
  OPEN(DSK, DEVICE, 15, 'N' + S);
  CLOSE(DSK)
END; (* HEADER *)
```

Beispiele:

```
HEADER('NEU',8);
HEADER('DATEN,XX',9);
```

HEX$

```
FUNCTION HEX(X:INTEGER): STRING;
(* wandelt X in eine vierstellige Hexadezimalzahl um *)
  FUNCTION HEXBYTE(B:BYTE): STRING;
    FUNCTION HEXDIGIT(D: BYTE): CHAR;
```

```
    BEGIN
      IF D>9 THEN HEXDIGIT:= CHR(D - 10 + ORD('A'))
             ELSE HEXDIGIT:= CHR(D + ORD('0'));
    END; (* HEXDIGIT *)
  BEGIN
    HEXBYTE:= HEXDIGIT(B DIV 16) + HEXDIGIT(B AND 15);
  END; (* HEXBYTE *)
BEGIN
  HEX:= HEXBYTE(HBYTE(X)) + HEXBYTE(X AND 255);
END; (* HEX *)
```

Beispiele:

```
WRITELN(HEX(1234));
WRITELN(HEX(-30));
```

KEY

Zur Definition von Tastaturbelegungen existiert im Betriebssystem bereits eine Routine, die in der folgenden Prozedur verwendet wird:

```
PROCEDURE KEY(NUMMER: BYTE; S:STRING);
(* Belege die Funktionstaste NUMMER mit dem String S *)
(* Nummer 1 bis 8 sind die Funktionstasten F1-F8     *)
(* Nummer 9 ist die SHIFT/RUN-STOP-Taste, während    *)
(* Nummer 10 die HELP-Taste bezeichnet               *)
  VAR REGS: RECORD SR,A,X,Y: BYTE; END;
      POINTER: RECORD
                 P: INTEGER;
                 BNK: BYTE
               END [252];
BEGIN
  POINTER.P:= ADDU(ADR(S),1);
  (* Zeiger auf 1. Zeichen des Strings in Bank 1 *)
  POINTER.BNK:= 1;
  REGS.A:= 252;      (* Zeiger auf Pointer in Z-Page  *)
  REGS.X:= NUMMER;
  REGS.Y:= LENGTH(S);
  BANK:=15; SYS(-155,REGS);
END; (* KEY *)
```

Beispiele:

```
KEY(1,'Dies ist F1');
KEY(2,'*' #13);
KEY(10,'Hilflos');
```

RECORD

Dieser Befehl entspricht der Prozedur SEEK, die im Programm in Bild 105 verwendet wurde. Die Anwendung relativer Dateien in *Pascal 2.0* wird in Kapitel 3.1 ausführlich im Zusammenhang erläutert.

RENAME

```
PROCEDURE RENAME(A,B:STRING; DEVICE:BYTE);
(* Benenne die Datei A in B um                    *)
  VAR DSK: TEXT;
BEGIN
  OPEN(DSK, DEVICE, 15, 'R' + B + ' =' + A);
  CLOSE(DSK)
END; (* COPY *)
```

Datei A und B müssen die gleiche Laufwerkkennzeichnung besitzen.
Beispiel:

```
RENAME('0:DATEN', '0:DATENNEU',8)
```

XOR

```
FUNCTION XOR(A,B:INTEGER): INTEGER;
(* Diese Funktion verknüpft die Zahlen A und B logisch *)
(* EXKLUSIV ODER                                       *)
BEGIN
  XOR:= (A AND NOT B) OR (NOT A AND B);
END; (* XOR *)
```

Die logische EXCLUSIV-ODER-Verknüpfung boolescher Werte A und B erfolgt mit dem Vergleich A< >B.

3.4 Automatisches Nachladen von Programmteilen

Programme länger als 64 Kbyte

Obwohl Sie für übersetzte Pascal-Programme die gesamte Bank 0 zur Verfügung haben, kann es notwendig werden, Programme zu schreiben, die länger als 64 Kbyte werden. Die sinnvollste Lösung für solche Anwendungen besteht darin, das Programm in mehrere Teile zu zerlegen, die möglichst unabhängig voneinander sind. Diese Teile kommunizieren dabei nur über globale Variablen. Beim Übergang von einem Teil zum anderen wird das entsprechende Teilprogramm von der Diskette nachgeladen.
Diese eigentlich triviale Idee beinhaltet jedoch einige Probleme: Nachdem das Folgeprogramm nachgeladen wurde, muß die Programmkontrolle explizit an das neue Programm übergeben werden. Außerdem muß dafür gesorgt werden, daß die globalen Variablen bei allen Programmen an den gleichen Adressen

gespeichert werden, damit sie zum Austausch von Informationen zwischen den Teilen verwendet werden können.

Allgemeingültige Methode zum Nachladen von Programmteilen

In diesem Kapitel wird eine Methode zum Nachladen von Programmen vorgestellt, die universell verwendbar ist. Zwar sind dabei einige manuelle Nacharbeiten erforderlich, jedoch sollten Sie bedenken, daß z.B. der Compiler *Pascal 2.0* vollständig in Pascal geschrieben ist und nur 22 Kbyte Speicherplatz benötigt. Sie müßten also Programme entwickeln, die mehr als doppelt so komplex sind wie der Compiler, bevor Sie die Speicherkapazität des C128 überfordern.

Die typische Struktur eines solchen großen Programmes zeigt Bild 108. Aus einem relativ kleinen Hauptprogramm werden viele Prozeduren aufgerufen, die weitgehend unabhängig voneinander sind.

```
PROGRAM RIESIG (INPUT, OUTPUT);
VAR A: ARRAY[0..100] OF INTEGER;
    WAHL: CHAR;
PROCEDURE VORBELEGEN;
  VAR I: INTEGER;
BEGIN
  FOR I:= 0 TO 100 DO A[I]:= I;
END; (* VORBELEGEN *)

PROCEDURE LOESCHEN;
  VAR I: INTEGER;
BEGIN
  FOR I:= 0 TO 100 DO A[I]:= 0;
END; (* LOESCHEN *)

PROCEDURE AUSGEBEN;
  VAR I: INTEGER;
BEGIN
  FOR I:= 0 TO 100 DO
    WRITE(A[I]:4);
  WRITELN;
END; (* AUSGEBEN *)

BEGIN
  REPEAT
    WRITELN('V)orbelegen  L)oeschen A)usgeben oder E)nde?');
        READLN(WAHL);
```

Bild 108: Struktur des Gesamtprogrammes (Fortsetzung nächste Seite) ►

```
    CASE WAHL OF
     'V': VORBELEGEN;
     'L': LOESCHEN;
     'A': AUSGEBEN;
     'E': HALT;
     ELSE
    END; (* CASE *)
   UNTIL FALSE;
END.
```

Bild 108: Struktur des Gesamtprogrammes (Schluß)

Auslagern globaler Deklarationen in ein INCLUDE-File

Zunächst stellen Sie alle Konstanten, Typ-, Variablen- und Prozedurdeklarationen, die global von allen Programmteilen verwendet werden sollen, in einem eigenen **INCLUDE-File** zusammen. Als **erste** Prozedurdeklaration in diesem INCLUDE-File steht die Prozedur CHAIN, die den gegenseitigen Aufruf der Teilprogramme steuert:

```
PROGRAM RIESIG (INPUT,OUTPUT);
VAR A: ARRAY[0..100] OF INTEGER;
    WAHL: CHAR;
PROCEDURE CHAIN(S: STRING);
(* Folgeprogramm S aufrufen. Der Rücksprung muß explizit *)
(* mit einem Aufruf der Prozedur CHAIN erfolgen!         *)
   PROCEDURE LOAD(S: STRING; TARGETBANK: INTEGER);
   (* DIE PROZEDUR LÄDT DIE DATEI MIT DEM NAMEN S VON DER *)
   (* DISKETTE IN DIE ANGEGEBENE BANK. DIE LADEADRESSE    *)
   (* IST MIT DEM FILE GESPEICHERT WORDEN. DIE PROZEDUR   *)
   (* ENTSPRICHT ALSO DEM BASIC-BEFEHL BLOAD"...",ONBx    *)
     VAR I   : INTEGER;
         REGS: RECORD
                 SR,AKKU,X,Y: BYTE
               END;

   BEGIN
     BANK:=15;
     WITH REGS DO
       BEGIN
        AKKU:= TARGETBANK; X:=1;        (* NAME IN BANK 1 *)
```

Bild 109: Inhalt des INCLUDE-Files GLOBAL.INC

```
        SYS(-152,REGS);                  (* SETBNK          *)
        AKKU:= LENGTH(S); I:= ADDU(ADR(S),1);
        X:= I; Y:= HBYTE(I);             (* ADRESSE NAME    *)
        SYS(-67, REGS);                  (* SETNAM          *)
                                         (* A = laden       *)
        AKKU:=0; X:=8; Y:=1;             (* X = Gerät       *)
                                         (* Y = absolut     *)
        SYS(-70,REGS);                   (* SETLFS          *)
        AKKU:=0; SYS(-43,REGS);          (* LOAD            *)
        IF ODD(SR) THEN                  (* CARRY=1 ?       *)
          BEGIN WRITELN('LADEFEHLER!'); HALT; END;
      END;
   END; (* LOAD *)

BEGIN
  LOAD(S,0); BANK:=0; SYS(7200);
END; (* CHAIN *)
```

Bild 109: Inhalt des INCLUDE-Files GLOBAL.INC (Schluß)

Zerlegung in Teilprogramme kürzer als 64 Kbyte

Die Operationen werden nun jeweils in Form eines eigenständigen Programmes definiert. Dabei werden die globalen Deklarationen von dem INCLUDE-File GLOBAL.INC gelesen.

```
(* HAUPTROGRAMM RIESIG.P *)
(*$"GLOBAL.INC" *)
BEGIN
  REPEAT
    WRITELN('V)orbelegen  L)oeschen A)usgeben oder E)nde?')
    READLN(WAHL);
    CASE WAHL OF
     'V': CHAIN('VORBELEGEN.M');
     'L': CHAIN('LOESCHEN.M');
     'A': CHAIN('AUSGEBEN.M);
     'E': HALT;
     ELSE
    END; (* CASE *)
  UNTIL FALSE;
END.
```

Bild 110: Zerlegung in Teilprogramme (Fortsetzung nächste Seite)

```
(* TEILPROGRAMM VORBELEGEN.P *)
(*$"GLOBAL.INC" *)
PROCEDURE VORBELEGEN;
  VAR I: INTEGER;
BEGIN
  FOR I:= 0 TO 100 DO A[I]:= I;
END; (* VORBELEGEN *)

BEGIN
  VORBELEGEN; CHAIN('RIESIG.M');
END.

(* TEILPROGRAMM LOESCHEN.P *)
(*$"GLOBAL.INC" *)
PROCEDURE LOESCHEN;
  VAR I: INTEGER;
BEGIN
  FOR I:= 0 TO 100 DO A[I]:= 0;
END; (* LOESCHEN *)

BEGIN
  LOESCHEN; CHAIN('RIESIG.M');
END.
(* TEILPROGRAMM AUSGEBEN.P *)
(*$"GLOBAL.INC" *)
PROCEDURE AUSGEBEN;
  VAR I: INTEGER;
BEGIN
  WRITELN('Die Daten lauten:');
  FOR I:= 0 TO 100 DO
    WRITE(A[I]:4);
  WRITELN;
END; (* AUSGEBEN *)

BEGIN
  AUSGEBEN; CHAIN('RIESIG.M');
END.
```

Bild 110: Zerlegung in Teilprogramme (Schluß)

Wenn Sie nun die in Bild 110 aufgeführten Teilprogramme übersetzen, so besitzen Sie anschließend vier Programmdateien auf der Diskette. Für jede Datei meldet Ihnen der Compiler die Endadresse der Konstanten.

```
RIESIG.M          Stack-Anfang = 8282
VORBELEGEN.M      Stack-Anfang = 8212
LOESCHEN.M        Stack-Anfang = 8212
AUSGEBEN.M        Stack-Anfang = 8229
```

Manuelle Modifikation des Stackanfangs bei allen Teilprogrammen

Bei dieser Adresse beginnt der **Variablenspeicher** der Programme. Da Sie natürlich in allen Programmen die in GLOBAL.INC deklarierten Konstanten und Variablen verwenden wollen, müssen Sie manuell in den Teilprogrammen die Anfangsadresse für den Variablenspeicher festlegen. Den meisten Platz für Konstanten belegt das Programm RIESIG.M, daher dürfen die Variablen in den anderen Programmen auch erst an der Adresse 8282 beginnen. Dazu laden Sie jeweils die Programme VORBELEGEN.M, LOESCHEN.M und AUSGEBEN.M. Mit folgenden BASIC-Befehlen modifizieren Sie die Anfangsadresse des *stack*:

```
CLR
A=PEEK(7397)+256*PEEK(7398)
B= 8282
POKE A,B AND 255
POKE A+1, B / 256
```

Wenn Sie anschließend die Programme wieder mit SAVE auf Diskette speichern, können Sie das Gesamtprogramm RIESIG.M mit RUN starten. Falls Sie eine Floppy des Typs 1571 besitzen, halten sich die Ladezeiten auch bei großen Programmen in erträglichem Rahmen.

Kapitel 4
Dokumentation Pascal-System

Files auf der Systemdiskette

Diesem Buch liegt eine Diskette im 1541-Format bei. Sie kann von allen an den C128 anschließbaren Diskettenlaufwerken gelesen werden. Vor der Arbeit mit dem Compiler sollten Sie sich jedoch eine Arbeitskopie der Systemdiskette anlegen und die Systemdiskette mit den Originalprogrammen an einem sicheren Platz aufheben. Das Anfertigen einer Arbeitskopie ist in Kapitel 1.3.1 ausführlich beschrieben.

Auf der Diskette finden Sie folgende Dateien:

PASCAL	PRG	Mit diesem Programm laden Sie das komplette Pascal-System in den Arbeitsspeicher des Rechners. Außerdem können Sie mit diesem Programm Arbeitskopien der Systemdiskette anlegen und die Dateien ERRORS.TXT und PAS.LIB kopieren.
SYSTEM.M COMPILER.M EDITOR.M	PRG PRG PRG	Diese Dateien enthalten den Code für das System und werden von dem Programm PASCAL geladen
PAS.LIB	PRG	Diese Datei enthält das Laufzeitsystem für übersetzte Pascal-Programme. Sie muß sowohl auf der Systemdiskette als auch auf jeder Diskette, die übersetzte Pascal-Programme enthält, vorhanden sein. Eine Kopie dieser Datei können Sie mit dem Programm PASCAL erstellen.
ERRORS.TXT	SEQ	Diese Datei enthält zu jeder Fehlernummer des Compilers einen erläuternden Text (in englischer Sprache). Sollten Sie einmal nicht genug Platz

		auf der Arbeitsdiskette besitzen, so können Sie diese Datei löschen (natürlich **nicht auf der Originaldiskette!**)
xxxxxxxx.P	PRG	Pascal-Demonstrationsprogramme im Quelltext.

Zusätzliche Informationen

Sollten nach der Drucklegung des Buches Fehler im Programm oder in der Dokumentation festgestellt werden, so finden Sie eine Datei mit dem Namen BITTE.LESEN auf der Diskette. Diese können Sie mit dem Editor wie einen normalen Quelltext lesen.

Start des Systems

Zunächst müssen Sie alle Erweiterungsmodule abschalten, da diese eventuell die Funktion des Pascal-Systems stören könnten. Jetzt laden Sie das Programm PASCAL von der Systemdiskette und starten es:

```
LOAD "PASCAL",8
RUN
```

Es erscheint folgendes Installationsmenü:

```
              PASCAL-SYSTEM
              =============
 1. System starten
 2. Fehlermeldungen kopieren
 3. "PAS.LIB" kopieren
 4. Systemdiskette kopieren
 5. System neu konfigurieren
 6. Abbruch (Ausgang zu BASIC)
 ==>1
```

Bild III: Installationsmenü des Pascal-Systems

Falls Sie bereits eine Arbeitskopie des Pascal-Systems erstellt haben, können Sie direkt durch die Betätigung der Return-Taste den Menüpunkt 1 (Start des Systems) wählen. Damit erreichen Sie das Pascal-Menü. Die übrigen Punkte dienen der Installation des Systems auf einer anderen Diskette und werden in Kapitel 1.3.1 beschrieben.

Allgemeine Hinweise zur Bedienung des Systems

Einheitliche Menüstruktur

Die Teilprogramme des Pascal-Systems besitzen eine einheitliche Menüstruktur. Alle Eingaben sind so organisiert, daß Sie mit möglichst wenigen Zwischenschritten jede Funktion erreichen können. Dabei besitzen im allgemeinen die Zeichen »*« und »?« eine Sonderfunktion.

Alle Eingaben bei blinkendem Cursor müssen mit der Return-Taste beendet werden. Grundsätzlich wird bei längerer Operation (Laden, Speichern, Kompilieren) in kurzen Zeitabständen eine Abfrage der Run/Stop-Taste vorgenommen, so daß die Ausführung jederzeit unterbrochen werden kann. Außerdem kann **jederzeit** mit der Run/Stop- und der Restore-Taste ein laufendes Programm abgebrochen werden. In diesem Fall erfolgt eine Rückkehr zum Pascal-Menü. Beim Laden und Speichern auf Diskette kann ein solcher sofortiger Abbruch zum Verlust von Daten führen.

Laufendes Programm abbrechen durch Run/Stop und Restore

4.1 Das Pascal-Menü

```
PASCAL-MENU
-----------

SELECT OPTION:

NAME    EDIT NEW DATASET
'?'     RESUME EDIT
'$'     COMPILE DATASET
'@'      SHOW DIRECTORY
'*'     EXIT TO BASIC

==>
```

Bild 112: Das Pascal-Menü

Aus diesem Menü erreichen Sie alle Teile des Pascal-Systems. Es sind dabei folgende Eingaben zulässig:

Eingabe eines Namens

Quelltext laden

Durch die Eingabe eines Namens lädt man einen Quelltext von der Diskette in den Arbeitsspeicher. Der Name darf maximal 16 Zeichen umfassen und darf die Sonderzeichen »*« und »?« nicht enthalten.
Der Editor sucht einen Text auf der Diskette im Laufwerk 0 mit der Geräteadresse 8. Als Typ wird das Suffix »PRG« benutzt. Tritt beim Ladevorgang ein Fehler auf, so springt der Editor zum Pascal-Menü zurück. Überprüfen Sie in diesem Fall, ob die Floppy betriebsbereit war und eine formatierte Diskette eingelegt wurde.

Name auf Diskette nicht gefunden

Konnte der angegebene Text auf der eingelegten Diskette nicht gefunden werden, so nimmt der Editor an, daß Sie einen neuen Text mit diesem Namen anlegen möchten. Es erscheint daher folgende Meldung:

```
THIS IS A NEW DATASET!

ENTER LINE LENGTH:

[RANGE: 1..136]
['*' OR '?' RETURN TO MENU]

==>
```

Bild 113: Anlegen eines neuen Quelltextes

Maximale Länge einer Textzeile = 136 Zeichen

Im Editor werden alle Texte mit einer festen Zeilenlänge gespeichert. Daher müssen Sie an dieser Stelle die maximale Länge einer Textzeile festlegen. Gültige Werte liegen im Bereich von einem bis 136 Zeichen pro Zeile. Jedoch bedeutet eine extrem große Zeilenlänge eine gewisse Speicherplatzverschwendung, da unnötig viele Leerstellen gespeichert werden. 60 Zeichen pro Zeile stellen eine sinnvolle Zeilenlänge dar.

Rückkehr zum Pascal-Menü

Hatten Sie sich jedoch nur bei der Namenseingabe im Pascal-Menü vertippt oder die falsche Diskette eingelegt, so können Sie mit der Eingabe von »*« oder »?« zum Pascal-Menü zurückkehren, ohne daß ein neuer Text angelegt wird.

Eingabe »?«

Daten editieren

Um einen Datenbestand zu editieren, der sich bereits im Arbeitsspeicher befindet, genügt diese Eingabe. Sie ersparen sich damit die Ladezeit von der Diskette. Dies ist insbesondere für schnelle Wechsel zwischen Editor und Compiler nützlich.

Eingabe »*«

Pascal-System verlassen

Mit diesem Befehl verlassen Sie das Pascal-System. Die nachfolgenden Tastatureingaben werden vom BASIC-Interpreter ausgewertet (siehe Abschnitt 4.3.3). Von BASIC gelangen Sie mit der nochmaligen Eingabe »*« zum Pascal-Menü zurück.

Eingabe "$"

Compiler aufrufen

Es wird der Compiler *Pascal 2.0* aufgerufen, der den Quelltext im Arbeitsspeicher übersetzt (siehe Abschnitt 4.3).

Eingabe "@"

Inhaltsverzeichnis aufrufen

Um das Inhaltsverzeichnis der eingelegten Diskette anzuzeigen, genügt die Eingabe des Klammeraffen-Zeichens »@«. Haben Sie die DIN-Tastaturbelegung (durch Drücken der DIN-Taste) gewählt, so wird anstelle des Klammeraffen das Paragraphenzeichen »%« erwartet und angezeigt.

Durch die Eingabe eines Suchmusters können Sie die anzuzeigenden Dateien genauer spezifizieren:

```
@DEMO*      (alle Files, die mit DEMO beginnen)
@DEMO???.P (alle Files,  die mit DEMO beginnen und mit .P enden)
@*=SEQ      (alle sequentiellen Files)
```

Nähere Angaben über den Aufbau von Suchmustern finden Sie im BASIC-Handbuch in Kapitel 6.5.

Längere Liste auf Bildschirm anhalten

Sollte die Anzeige länger als eine Bildschirmseite werden, kann sie mit der NOSCROLL-Taste angehalten werden. Nach der Anzeige des Verzeichnisses wird auf die Betätigung der Return-Taste gewartet, bevor das Pascal-Menü erneut angezeigt wird.

4.2 Der Editor

4.2.1 Arbeitsweise des Editors

Eingeben des Quelltextes

Um den Quelltext eines Pascal-Programmes einzugeben, ist im Pascal-System ein eigenständiges Textverarbeitungsprogramm enthalten. Mit ihm werden Texte im Arbeitsspeicher des Computers bearbeitet. Eingaben erfolgen über ein *Textfenster*, das in allen vier Richtungen (nach oben, unten, rechts und links) wie über ein Blatt Papier bewegt werden kann.

Damit die Texte nach dem Ausschalten erhalten bleiben, müssen sie auf Diskette gespeichert werden. Dies geschieht entweder automatisch beim Verlassen des Editors (mit dem Kommando END) oder durch den Editor-Befehl SAVE. Neben dem eigentlichen Text werden auch Informationen über Tabulatoren etc. gespeichert, die somit individuell für jeden Text eingestellt werden können.

Arbeit mit Bildschirm und Monitor

Aus drucktechnischen Gründen zeigen alle Bildschirmausdrucke die Bilder, die sich bei der Verwendung im 40-Zeichen-Modus (z.B. beim Anschluß eines Fernsehempfängers am HF-Ausgang) ergeben. Der Editor unterstützt jedoch auch den 80-Zeichen Modus, wobei sogar der gleichzeitige Betrieb zweier Bildschirme möglich ist.

Länge des Quelltextes

Innerhalb des Pascal-Systems stehen Ihnen über 22 Kbyte Textspeicher zur Verfügung. Durch den Einschluß von Programmtexten, die auf der Diskette gespeichert vorliegen, können jedoch beliebig große Quelltexte vom Compiler verarbeitet werden. Einzelheiten über die Verwendung dieser sogenannten INCLUDE-Files finden sich in Abschnitt 4.4.6.4.

4.2.2 Gliederung des Bildschirms

Bei der Arbeit mit dem Editor ist der gesamte Bildschirm (wie ein Formular) in verschiedene Bereiche eingeteilt, die jeweils spezielle Aufgaben besitzen. Mit den Cursortasten können Sie sich frei über den gesamten Bildschirm bewegen, um gezielt in den einzelnen Feldern Texte oder Kommandos einzugeben.

```
1.COL:0000                    SCROLL:HALF
 COL=0----+----1----+----2----+----3----+
    **************** TOP ***************
0000 PROGRAM PROGRAMM1(OUTPUT);
0001
0002   VAR I,N : INTEGER;
0003       R   : REAL;
0004
0005 BEGIN
0006  WRITE('N='); READLN(N);
0007  FOR I:= N TO 2*N DO
0008    BEGIN
0009      R:= 1/I;
0010      WRITELN(I:3, R:15);
0011    END;
0012 END.
0013
0014
    ************** BOTTOM **************
```

Bild 114: Der Aufbau des Bildschirms im Editor

Kommandoeingaben

Bei der gleichzeitigen Betätigung der Tasten <Shift> und <Return> wird der gesamte Bildschirm ausgewertet und die eventuell vorhandenen Kommandos ausgeführt. Somit ist es insbesondere möglich, an verschiedenen Stellen des Bildschirms Kommandos einzugeben, die bei der Eingabe von <Shift> und <Return> **gleichzeitig** ausgeführt werden.
Konkret könnten Sie zum Beispiel im Textbereich Veränderungen durchführen, einen Zeilenblock löschen, einige Zeilen einfügen und gleichzeitig den Scroll-Betrag verändern.
Die Namen und die Bedeutung der einzelnen Bereiche gehen aus den folgenden Beschreibungen hervor. Die Namen Kopfzeile, Scroll-Betrag, Statuszeilen, Zeilennummernbereich und Textfenster werden im gesamten Buch zur eindeutigen Identifikation der Bereiche verwendet.

4.2.2.1 Die Kopfzeile

Meldungen und Kommandoeingabe

In diesem farblich hervorgehobenen Bereich werden (Fehler-) Meldungen des Editors angezeigt. Liegen keine Meldungen vor, so wird die Nummer der ersten Textspalte im Textfenster angezeigt.
`1.COL:0041`

bedeutet, daß der erste Buchstabe am linken Rand des Textfensters im Text in der Spalte 41 steht. Der Bildschirm ist also um 40 Spalten nach rechts verschoben worden.
Andererseits wird die Kopfzeile auch zur Eingabe von Befehlen, der sogenannten *primary-commands* (siehe Abschnitt 4.2.4) verwendet. Befindet sich der Cursor bei der Eingabe in der Kopfzeile, so wird eine zuvor angezeigte Meldung automatisch ausgeblendet.

4.2.2.2 Der Scroll-Betrag

Blätterschrittweite

Beim horizontalen oder vertikalen Blättern (engl. *scrolling*) durch den Text verschiebt sich das Textfenster um einen frei wählbaren Betrag, der ebenfalls in der ersten Bildschirmzeile angezeigt wird. Sie können diesen Wert jederzeit durch Überschreiben ändern. Dabei sind folgende Spezifikationen erlaubt:

HALF Beim horizontalen Blättern verschiebt sich das Fenster um eine halbe Bildschirmbreite, beim vertikalen Blättern um eine halbe Bildschirmhöhe.

PAGE Beim horizontalen Blättern verschiebt sich das Fenster um eine ganze Bildschirmbreite, beim vertikalen Blättern um eine ganze Bildschirmhöhe.

nnnn Eine vierstellige Zahl kleiner als 128. Bei jedem Blättern verschiebt sich das Fenster um nnnn Spalten bzw. Zeilen.

Voreingestellt ist der Wert HALF.

Beispiel:

Bewegen Sie den Cursor an die Position des Buchstabens H im Wort HALF und überschreiben Sie ihn mit der Ziffer »1«. Nach der Auswertung der Eingabe (<Shift> <Return> eingeben), wird anschließend in Einerschritten geblättert.

4.2.2.3 Die Statuszeilen

Statuszeilen im Text auf Diskette gespeichert

Diese Zeilen werden nur auf Wunsch, z.B. nach dem *primary-command* PROFILE zwischen der Kopfzeile und dem Textfenster angezeigt. Sie können auch selektiv eingeblendet oder wieder ausgeblendet werden. Der Inhalt dieser Zeilen wird mit dem Text auf Diskette gespeichert, so daß Änderungen dieser Zeilen individuell für jeden Text vorgenommen werden können.
Bis auf die Spaltenmarkierungen können alle Zeilen jederzeit durch Überschreiben verändert werden.

Einfügemaske

Die Zeile nach *MSK=* bildet eine Maske, die beim Einfügen neuer Zeilen vorgegeben wird. Innerhalb der Maske sind alle von der Tastatur erreichbaren Zeichen erlaubt. Beim Start des Editors besteht die Maske aus einer Leerzeile.

Beispiel:

Durch die Eingabe von (*) am rechten Rand der Maske wird bei späteren*

Einfügeoperationen (line-command I) in jeder Zeile eine öffnende und schließende Kommentarklammer vorgegeben.

Textgrenzen

In der Zeile hinter *BND=* wird der Spaltenbereich begrenzt, der für die Texteingabe verwendet werden soll. Auch die Befehle FIND und CHANGE und das »überlappende« Kopieren (mit dem *line-command* O) wird auf diesen Spaltenbereich begrenzt. Gültige Zeichen, die jeweils nur genau einmal auftreten dürfen, sind:

> markiert die letzte Spalte, in der Text eingegeben wird.

< markiert die erste Spalte, in der Text eingegeben wird.

Beim Start des Editors steht < in der ersten Textspalte, während > in der letzten Textspalte steht.

Beispiel:

Bei der Eingabe eines mehrspaltigen Textes (z.B. einer Dokumentation) können die Textgrenzen verändert werden, um Einfüge- und Suchoperationen auf den angegebenen Bereich in der Zeile zu begrenzen.

Tabulatoren

In der Zeile nach *TAB=* können Tabulatoren gesetzt werden. Jedes Zeichen außer dem Leerzeichen wirkt als ein gesetzter Tabulator, der beim nächsten Bildaufbau als ein Strich (»–«) dargestellt wird. Tabulatoren werden im Textmodus (siehe Abschnitt 4.2.6) mit der Taste <Shift> <Return> angesprungen. Am Beginn der Editierung sind alle Tabulatoren gelöscht.

Spaltenmarkierungen

Schließlich wird nach *COL=* eine Spaltenmarkierung ausgegeben, die zur Orientierung im Text dient: An jeder zehnten Position wird eine Dezimalziffer gedruckt, nach fünf Schritten noch ein Pluszeichen.

Diese letzten beiden Zeilen sind insbesondere bei der übersichtlichen Einrückung von Pascal-Programmen nützlich, um z.B. zu einem BEGIN das schließende END anhand der »Einrückungstiefe« zu finden.

4.2.2.4 Die Zeilen TOP und BOTTOM

Textanfang und -ende

Diese beiden Zeilen kennzeichnen am Bildschirm den Anfang und das Ende des Textes im Arbeitsspeicher. Zu beachten ist die Tatsache, daß man **vor** der ersten Zeile Text einfügen kann, indem man in der Zeile TOP (vor den Sternen) das Zeilenkommando (*line-command*) I eingibt.

4.2.2.5 Der Zeilennummernbereich

Automatische Numerierung zur Orientierung

Alle Textzeilen sind vierstellig von 0000 bis 9999 aufsteigend durchnumeriert. Die Zeilennummern dienen nur zur Orientierung im Text und werden nicht mit dem Text gespeichert. Beim Löschen und Einfügen von Zeilen werden die nachfolenden Zeilen automatisch neu numeriert, so daß der gesamte Text ständig in Einerschritten numeriert bleibt.

line-commands

Im farblich hervorgehobenen Zeilennummernbereich können ebenfalls Kommandos, die sogenannten *line-commands* (Zeilenkommandos, siehe Abschnitt

4.2.5) eingegeben werden. Dies erreicht man, indem man die Zeilennummern mit den entsprechenden Zeichen (D, M, C etc.) überschreibt. Unvollständige Blockkommandos (z.B. ») werden gespeichert und bleiben an ihrer alten *Textposition*, auch wenn inzwischen durch Einfügungen oder Löschungen ein Umnumerieren erfolgt ist.

4.2.2.6 Das Textfenster

Texteingabe nur im Textfenster möglich

Die eigentliche Texteingabe erfolgt in dem sichtbaren Textausschnitt, der auf dem Bildschirm durch folgende Grenzen bestimmt wird: Das Textfenster beginnt unterhalb der Kopfzeile bzw. der Zeile TOP und endet am unteren Bildschirmrand oder der Zeile BOTTOM. Beim Anlegen eines Textes wurde die maximale Länge einer Zeile definiert. Eine Eingabe längerer Zeilen ist nicht möglich, da nachfolgende Spalten auf dem Bildschirm für Eingaben von der Tastatur gesperrt sind.

4.2.3 Cursorsteuerung

Sondertasten

Der Cursor wird mit den folgenden Tasten gesteuert, die – soweit möglich – dieselben Funktionen wie im BASIC-Editor besitzen:

<CRSR UP>	bewegt den Cursor auf dem Bildschirm eine Zeile höher. In Zeile 1 erfolgt ein Sprung in die letzte Zeile.
<CRSR DOWN>	bewegt den Cursor auf dem Bildschirm eine Zeile tiefer. In Zeile 25 erfolgt ein Sprung in die erste Zeile.
<CRSR -> >	bewegt den Cursor eine Spalte nach rechts. Am rechten Bildrand erfolgt ein Sprung in die erste Spalte der gleichen Zeile. Im Textmodus (siehe Abschnitt 4.2.6) folgt der Textausschnitt der Bewegung des Cursors nach rechts, falls er sich innerhalb des Textfensters befand.
<CRSR <->	bewegt den Cursor eine Spalte nach links. Am linken Bildrand erfolgt ein Sprung in die letzte Spalte der gleichen Zeile.

Es können sowohl die Cursortasten des C64-Modus als auch die abgesetzten Pfeiltasten benutzt werden.

<Home>	Der Cursor wird in die linke obere Bildschirmecke gesetzt. Befand sich der Cursor bereits an der ersten Position der Kopfzeile, so springt der Cursor zur ersten Position im Scroll-Betrag.
<CLR>	Die Kopfzeile wird gelöscht und der Cursor wird an die erste Position der Kopfzeile gesetzt.
<Inst>	Diese Taste arbeitet nur in der Kopfzeile, den Statuszeilen und im Textfenster. An der momentanen Cursorposition wird ein Leerzeichen eingefügt. Der Text bis zur letzten Textspalte (Markierung > in der Zeile BND=) wird nach rechts verschoben. Ist am Zeilenende kein Platz mehr, so ist die Inst-Taste gesperrt.
<Del>	Diese Taste ist ebenfalls nur in der Kopfzeile, den Statuszeilen und im Textfenster aktiviert. Das Zeichen links vom Cursor wird gelöscht, und der Text bis zur letzten Textspalte (Markierung > in der Zeile BND=) wird nach links verschoben.

<Return>	Die Wirkung dieser Taste ist abhängig vom momentan gültigen Eingabemodus. Befindet sich der Cursor im Textfenster und ist der Textmodus aktiviert, so erfolgt ein Sprung zur ersten Textspalte (Markierung < in der Zeile BND=) der nachfolgenden Textzeile. Dabei folgt das Textfenster der Bewegung des Cursors. Ansonsten springt der Cursor zur ersten Bildschirmspalte der Folgezeile. In der letzten Bildschirmzeile springt der Cursor zur ersten Bildschirmzeile.
<Shift><Return>	Diese Tastenkombination wertet den aktuellen Bildschirm aus. Alle Kommandos werden gelesen und ausgeführt. Im Textmodus erfolgt außerdem ein Sprung zum nächsten gesetzten Tabulator in der Zeile. Ist kein Tabulator gesetzt, so springt der Cursor an den Textanfang (Markierung < in der Zeile BND=) der Folgezeile.

Statt der Taste <Return> kann auch alternativ die Taste <Enter> beim Ziffernblock betätigt werden. Die folgenden Funktionstasten bewegen generell (nicht nur im Textmodus) den gesamten Bildschirmausschnitt.

Blättern im Text

Befindet sich der Cursor beim Blättern im Textfenster, so bleibt er auf seiner Textposition, falls diese auf dem folgenden Bild noch vorhanden ist, ansonsten springt er in die linke obere Bildschirmecke. Befindet sich der Cursor außerhalb des Textfensters, so bleibt er auf seiner Bildschirmposition.

<F1> Das Textfenster wird nach oben um den gewählten Scroll-Betrag verschoben. Dadurch werden Zeilen mit niedrigeren Zeilennummern sichtbar.

<F3> Das Textfenster wird um den gewählten Scroll-Betrag zum Textende hin verschoben.

<F5> Das Textfenster wird um den gewählten Scroll-Betrag nach rechts verschoben.

<F7> Das Textfenster wird um den gewählten Scroll-Betrag nach links (zum Zeilenanfang hin) verschoben.

Wiederholen der Suchfunktionen

<F2> Der letzte FIND-Befehl wird wiederholt (siehe Abschnitt 4.2.7). Der Cursor wird auf die entsprechende Textposition gesetzt.

<F4> Der letzte CHANGE-Befehl wird wiederholt (siehe Abschnitt 4.2.8). Der Cursor wird auf die geänderte Textstelle gesetzt.

Außerdem lassen sich noch die Tasten <Shift><Run/Stop>, <Help>, <F6> und <F8> mit Text- und Kommandosequenzen belegen. Nähere Einzelheiten hierzu finden Sie in Abschnitt 4.2.13.

Sonderzeichen im Textfenster

Alle anderen Sondertasten (<RVS ON>, <CTRL BLK>, <Esc>, <Alt>) werden nicht ausgewertet, sondern als Steuercodes angezeigt und auch in den Text eingefügt.

Beispiel:

Sie könnten z.B. in eine WRITE-Anweisung das Steuerzeichen für die Umschaltung auf invertierte Darstellung (RVS ON) aufnehmen. Das Zeichen »RVS ON« wird am Bildschirm als ein invertiertes »R« dargestellt:

```
WRITELN('R Dieser Text wird invers gedruckt');
```

Drucken Sie jedoch den Quelltext mit diesem Steuerzeichen, so wird der Drucker das Zeichen interpretieren und den Text im Listing invers drucken. Deshalb sollten Sie lieber folgende Eingabe verwenden, die darüber hinaus auch leichter zu lesen ist:

```
WRITELN(CHR(18), 'Dieser Text wird invers gedruckt');
```

4.2.4 Die »primary-commands«

Globale Kommandos

Diese globalen Kommandos beziehen sich auf den gesamten zu editierenden Text. Sie werden im Klartext, also nicht als verschlüsselte Tastenkombinationen in der Kopfzeile eingegeben. Die dort angezeigte Meldung wird dabei automatisch ausgeblendet. Viele Befehle lassen sich auf die ersten Buchstaben abkürzen, so daß Sie statt des Kommandos LOCATE auch folgende Abkürzungen verwenden können:

Abkürzungen

```
LOCAT, LOCA, LOC, LO, L
```

Die Befehle werden bei der Eingabe der Tastenkombination <Shift> <Return> oder automatisch bei jedem Blättern ausgeführt. Einige Kommandos erwarten weitere Parameter. Diese Parameter müssen durch mindestens ein Leerzeichen vom Kommandowort getrennt sein:

```
LOCATE 4              (richtig)
LOCATE     4          (richtig)
LOCATE4               (falsch)
```

Parameter

Sind Parameter nur optional, so werden sie in eckigen Klammern aufgeführt. Daher sind die beiden folgenden Kommandos korrekt:

```
SAVE      oder
SAVE 8
```

Wird ein Befehl eingegeben, der in der folgenden Tabelle nicht aufgeführt ist, so wird der Inhalt der gesamten Kopfzeile als ein Befehl an die momentan angemeldete Diskettenstation gesendet.

Senden von Befehlen an die Floppy

```
SCRATCH0:TEXT1.P
```

Falls Sie diesen Text in der Kopfzeile eingeben und mit <Shift> <Return> bestätigen, so wird an die angemeldete Diskettenstation das Kommando zum Löschen der Datei »TEXT1.P« im Laufwerk 0 geschickt. Wurde diese Datei gefunden, so wird sie gelöscht, und die Diskettenstation bestätigt die Aktion mit der Meldung

```
01, FILES SCRATCHED, 01, 00
```

Lesen des Floppy-Status

Die Meldungen der Diskettenstation haben immer das Format

```
Fehlermeldung, Fehlertext, Spur, Sektor
```

Die Beschreibung aller Diskettenkommandos und der möglichen Meldungen der Floppy finden Sie im BASIC-Handbuch in den Kapiteln 6.8 bis 6.95 und 8.2. Es folgt eine Auflistung aller verfügbaren *primary-commands*:

RESET Kurzform: RES

Rücksetzen unvollständiger Kommandos

Mit diesem Befehl werden alle Statuszeilen, die z.B. mit PROFILE eingeblendet wurden, wieder ausgeblendet. Außerdem werden alle unvollständigen Blockbefehle (*line-commands* wie z.B. eine einzelne DD-Markierung) aufgehoben. Der Textmodus (siehe Abschnitt 4.2.6) wird ausgeschaltet. Die Funktionstasten <F2> und <F4> (siehe Abschnitt 4.2.9) werden deaktiviert. Die mit REPEAT einschaltbare Wiederholfunktion für alle Tasten wird ausgeschaltet.

PROFILE Kurzform: PROF

Anzeige der Statuszeilen

Die Statuszeilen »MSK=«, »BND=«, »TAB=«, und »COL=« zwischen der Kopfzeile und dem Textfenster werden in dieser Reihenfolge eingeblendet.

MSKS Kurzform: MSK

Die Einfügemaske »MSK=« wird angezeigt (siehe Abschnitt 4.2.2.3).

BNDS Kurzform: BND

Eine Zeile mit den aktuell gültigen Textgrenzen wird angezeigt (siehe Abschnitt 4.2.2.3)

TABULATOR Kurzform: TAB

Eine Zeile mit den gesetzten Tabulatoren wird angezeigt (siehe Abschnitt 4.2.2.3)

COLUMNS Kurzform: COL

Direkt oberhalb des Textfensters wird eine Zeile mit Spaltenmarkierungen angezeigt. Die Markierungen beziehen sich nicht auf Bildschirm- sondern auf Textspalten, so daß sich die Markierungen beim horizontalen Blättern verändern.

END [Geräteadresse] Kurzform: END

Verlassen des Editors

Der Text wird wie nach dem *primary-command* SAVE gespeichert. Tritt beim Speichern kein Fehler auf, so wird der Editor verlassen. Sie erreichen danach das Pascal-Menü.

CANCEL Kurzform: CAN

Der Text wird *nicht* auf Diskette gespeichert. Sie verlassen den Editor und kehren zum Pascal-Menü zurück. Mit diesem Befehl können Sie schnell zwischen Editor, Compiler und Laufzeitsystem hin- und herschalten, ohne auf die langsame Diskettenstation zu warten. Jedoch sollten Sie in regelmäßigen Abständen den Text mit SAVE oder END sichern, um vor zu großen Datenverlusten durch Fehlbedienung des Systems geschützt zu sein.

SAVE [Geräteadresse] Kurzform: SAVE

Speichern des Textes mit VERIFY

Mit diesem Befehl wird der Inhalt des Arbeitsspeichers auf Diskette unter dem Namen gespeichert, der beim Eintritt in den Editor vergeben wurde.

Zunächst wird eine eventuell vorhandene Datei gleichen Namens auf der Diskette gelöscht. Anschließend wird der editierte Text zusammen mit den Statuszeilen auf der Diskette gespeichert. Zur Sicherheit wird anschließend ein Ver-

gleich zwischen den gespeicherten Daten und dem Inhalt des Arbeitsspeichers durchgeführt. Stimmen beide nicht überein, so werden die obigen Schritte wiederholt. Tritt beim Speichern eine Fehlermeldung der Diskettenstation auf, so wird das Speichern unterbrochen und die Fehlermeldung angezeigt.

```
26, WRITE PROTECT ON,18,10
```

signalisiert z.B., daß Sie den Schreibschutz der Diskette nicht entfernt hatten, während die Fehlermeldung

```
SYSTEM ERROR NR:5
```

darauf hindeutet, daß die Diskettenstation nicht eingeschaltet ist.

Arbeit mit mehreren Diskettenstationen

Wird eine ganze Zahl hinter SAVE oder END genannt, so handelt es sich dabei um die Geräteadresse einer Diskettenstation an der seriellen Schnittstelle, auf der die Daten gespeichert werden sollen. Fehlt eine solche Angabe, so werden die Daten auf der angemeldeten Diskettenstation (siehe *primary-command* DRIVE) gespeichert.

TEXT Kurzform: TE

Textmodus wählen

Mit diesem Befehl schalten Sie den Textmodus (siehe Abschnitt 4.2.6) ein. Beim Aufruf des Editors wird der Textmodus automatisch eingeschaltet.

REPEAT Kurzform: REP

Tastenwiederholung

Alle Tasten (auch die Funktionstasten) erhalten eine REPEAT-Funktion. Damit werden Tasten, die länger gedrückt gehalten werden, automatisch wiederholt.

LOWER Kurzform: LOW

Groß- und Kleinschreibung

Standardmäßig ist der Bildschirm innerhalb des Editors auf die Anzeige von Großbuchstaben und Grafikzeichen eingestellt. Alternativ können Sie jedoch mit diesem Befehl auf die Darstellung von Klein- und Großbuchstaben umstellen. Grundsätzlich müssen jedoch Namen und Befehle in Pascal-Programme *ohne* Betätigung der Shift-Taste eingegeben werden. Nur innerhalb von Kommentaren und Strings dürfen beliebig Groß- und Grafikzeichen verwendet werden.

UPPER Kurzform: UP

Die Bildschirmdarstellung wird wieder auf Großschrift und Grafikzeichen umgestellt. Ein Mischen der Darstellung in Kleinschrift und Grafikzeichen innerhalb eines Textes ist nicht möglich.

LOCATE [Zeilennummer] Kurzform: L

Sprung zu einer Zeile

Das Textfenster wird vertikal so verschoben, daß die Zeile mit der genannten Zeilennummer als erste Zeile am Bildschirm erscheint. Wird eine zu große Zeilennummer angegeben, so wird zum Ende des Textes (dort steht die Zeile BOTTOM) geblättert. Fehlt die Angabe einer Zeilennummer, so entspricht dies dem Kommando

```
LOCATE 0
```

Es wird also zum Textanfang (dort steht die Zeile TOP) geblättert.

DIRECTORY [Suchmuster] Kurzform: DIR

Inhaltsverzeichnis

Dieses primary-command zeigt den Inhalt der Diskette im angemeldeten Laufwerk an. Mit dem optionalen Suchmuster können Sie die anzuzeigenden Dateien näher spezifizieren. Die Regeln für den Aufbau des Suchmusters sind bei der Beschreibung des Befehls »@« im Pascal-Menü und im BASIC-Handbuch in Abschnitt 6.5 genannt. Ist die Ausgabe länger als eine Bildschirmseite, kann sie mit der NOSCROLL-Taste angehalten werden. Am Ende der Ausgabe wird auf die Betätigung einer Taste gewartet, bevor der Bildschirm im Editor neu aufgebaut wird.

SWAP Kurzform: SWA

Wahl des Anzeigemodus

Dieser Befehl ist der Schlüssel zur simultanen Verwendung eines 40- und eines 80-Zeichen-Monitors im Editor. Außerhalb des Editors können Sie jederzeit mit der Tastenfolge <Esc> und anschließend <X> von einem Bildschirm zum anderen schalten. Innerhalb des Editors erfolgt diese Umschaltung mit dem *primary-command* SWAP. Sie könnten somit theoretisch auf dem 40-Zeichen-Bildschirm einen Textausschnitt darstellen, während Sie auf dem 80-Zeichen-Bildschirm an einer anderen Textposition neue Eingaben vornehmen.

Beim C 128 ist die Anzeige im 40-Zeichen-Modus nur möglich, wenn die Taktfrequenz des Prozessors 1 MHz beträgt. Sollten Sie also beim Umschalten in den 40-Zeichen-Modus nur ein einfarbiges Bild sehen, können Sie den Editor mit CANCEL verlassen und in BASIC das Kommando SLOW eingeben.

Arbeiten Sie jedoch nur im 80-Zeichen-Modus, so sollten Sie den Rechner im FAST-Modus betreiben, da dadurch die Bildaufbauzeit halbiert wird.

DRIVE [Geräteadresse] Kurzform: DRIVE

Anmelden eines Laufwerks

Beim Start des Pascal-Systems wird zum Laden und Speichern standardmäßig auf die Diskettenstation mit der Geräteadresse 8 zugegriffen. Besitzen Sie jedoch mehrere Diskettenstationen mit verschiedenen Geräteadressen (8, 9, 10,...), so können Sie diese mit dem Befehl DRIVE, gefolgt von der Geräteadresse, anmelden. Anschließend beziehen sich alle Lade- und Speicheroperationen sowie die Diskettenbefehle, die in der Kopfzeile eingegeben werden können, auf dieses Laufwerk.

Andererseits können Sie bei den Operationen SAVE, END und COPY eine Geräteadresse nennen, die *nur für diese Operation* Gültigkeit besitzt. Das bedeutet, daß das angemeldete Laufwerk nicht verändert wird.

FIND Kurzform: F

Suchen

Mit diesem Befehl wird nach Zeichenfolgen im Text gesucht. Es ist eine Reihe von Parametern zulässig, die in Abschnitt 4.2.7 explizit beschrieben sind.

CHANGE Kurzform: C

Ersetzen

Mit diesem *primary-command* ist das gezielte Ersetzen von Zeichenfolgen durch andere Zeichenfolgen möglich. Dieser Befehl wird in einem separaten Abschnitt (4.2.8) beschrieben.

INPUT Kurzform: IN

Dieser Befehl erlaubt das Einfügen sequentieller Dateien von beliebigen Peripheriegeräten an einer frei wählbaren Textposition. Zur Ausführung des Kommandos (nach Betätigung von <Shift> <Return>) gelangen Sie in ein spezielles Menü, das in Abschnitt 4.2.10 beschrieben wird.

OUTPUT Kurzform: OUT

Sequentielle Files schreiben

Dieser Befehl ist das Gegenstück zum *primary-command* INPUT: Der Inhalt des Arbeitsspeichers wird als sequentielle Datei an ein Peripheriegerät geschickt. Wie bei dem Befehl INPUT kann die Ausgabe in einem eigenen Menü gesteuert werden (siehe Abschnitt 4.2.11).

COPY [Geräteadresse] Kurzform: COPY

Texte von Diskette lesen

Um Teile bereits erstellter Texte von der Diskette zu lesen, können Sie dieses Kommando verwenden. Die geladenen Teile können an einer beliebigen Stelle im Text eingefügt werden. Die Bedeutung des Parameters »Geräteadresse« wird ebenfalls in Abschnitt 4.2.12 beschrieben.

4.2.5 Die »line-commands«

Ein Buchstabe für jeden Zeilenbefehl

Viele Kommandos bei der Texteditierung beziehen sich auf spezielle Zeilen des Textes. Daher werden in diesem Editor solche Befehle im Zeilennummernbereich direkt vor der betreffenden Zeile eingegeben. Jeder Befehl wird durch einen Buchstaben oder ein Zeichen beschrieben. Die Buchstaben lehnen sich dabei an die englischen Namen für die auszuführende Operation an.

Beispiel:

Um die fünfte Zeile des Textes zu löschen, bewegen Sie den Cursor in den Zeilennummernbereich der Zeile 5. Dort geben Sie (»über« die Zahlen 0005) den Buchstaben »D« (für delete) ein.

Wie die *primary-commands* werden auch die *line-commands* erst bei der Eingabe der Tastenkombination <Shift> <Return> oder beim Blättern ausgeführt. Es dürfen mehrere verschiedene Kommandos in verschiedenen Zeilen eingegeben werden. Sie können also z.B. gleichzeitig eine Zeile einfügen und an einer anderen Stelle Zeilen löschen.

Blockbefehle

Einige Befehle können sich auch auf einen Block von Zeilen beziehen. In diesem Fall wird die erste und die letzte Zeile des Blockes mit einem doppelten Buchstaben im Zeilennummernbereich markiert. Ein Blockkommando wird erst ausgeführt, wenn der Anfang und das Ende des Blockes markiert wurden.

Beispiel:

Um die Zeilen 5, 6, 7 und 8 zu löschen, schreiben Sie zunächst in den Zeilennummernbereich der Zeile 5 das Kommando »DD«. Anschließend markieren Sie die letzte Zeile des Blockes (Zeile 8) im Zeilennummernbereich mit dem Kommando »DD«. Wenn Sie nun <Shift> <Return> *betätigen, werden alle Zeilen zwischen 5 und 8 (einschließlich) gelöscht.*

Wiederholungsfaktor

Bei einigen *line-commands* kann auch eine Zahl n angegeben werden, die angibt, wie oft eine Operation wiederholt werden soll.

Beispiel:

Der Befehl I5 im Zeilennummernbereich der Zeile 8 fügt hinter der Zeile 8 fünf Leerzeilen ein.

Die Befehle RR in Zeile 1 und RR8 in Zeile 2 wiederholen den Block, bestehend aus Zeile 1 und 2, insgesamt achtmal.

Ein Block darf nicht mehr als 256 Zeilen umfassen. Ohne Angabe eines Wiederholungsfaktors wird die Operation nur einmal ausgeführt. Es folgt eine Liste aller *line-commands*.

I n (Insert)

Nach der markierten Zeile werden n Zeilen eingefügt. Diese Zeilen werden mit dem Inhalt der Statuszeile nach »MSK=« (siehe Abschnitt 4.2.2.3) vorbelegt. Die nachfolgenden Textzeilen werden nach unten geschoben und neu numeriert.

Das *line-command* I ist auch in der Zeile TOP (links von den Sternen) erlaubt. An dieser Stelle bewirkt es das Einfügen *vor* der ersten Textzeile (mit der Nummer 0000).

D (Delete)

Die markierte Zeile wird gelöscht. Die nachfolgenden Zeilen werden nach oben geschoben und neu numeriert.

DD (Delete)

Der Block zwischen der ersten und der zweiten mit »DD« gekennzeichneten Zeile (einschließlich) wird gelöscht.

R n (Repeat)

Die markierte Zeile wird n-mal wiederholt.

RR n (Repeat)

Der durch zwei »RR«-Kommandos gekennzeichnete Block wird n-mal wiederholt.

> n (Shift right)

Der Inhalt der markierten Zeile wird um n Spalten nach rechts verschoben. Dabei gelten folgende Regeln: Es werden nur Texte zwischen den Markierungen < und > in der »BND=«-Statuszeile verschoben. Die restlichen Spalten bleiben unverändert. Eine Zeile wird *nicht* verschoben, wenn dadurch Zeichen am rechten Textrand (Markierung > in der Zeile »BND=«) herausgeschoben würden. In diesem Fall wird die Meldung »MOVE ERROR« angezeigt.

>> n (Shift right)

Die Funktionsweise entspricht dem *line-command* > für einen Block von Zeilen. Konnte eine Zeile des Blockes nicht verschoben werden, so wird ebenfalls die Meldung »MOVE ERROR« ausgegeben.

< n (Shift left)
Der Inhalt der markierten Zeile wird um n Spalten nach links verschoben. Es werden nur Texte zwischen den Markierungen < und > in der »BND=«-Statuszeile verschoben. Die restlichen Spalten bleiben unverändert. Eine Zeile wird nicht verschoben, wenn dadurch Zeichen am linken Textrand (Markierung < in der Zeile »BND=«) herausgeschoben würden. In diesem Fall wird die Meldung »MOVE ERROR« angezeigt.

< < n (Shift left)
Dieser Befehl bewirkt eine Verschiebung wie das *line-command* <, nur für einen Zeilenblock.

) n (Push right)
Der Inhalt der markierten Zeile wird um n Spalten nach rechts verschoben. Der Spaltenbereich wird ebenfalls durch die Einträge < und > in der Statuszeile nach »BND=« begrenzt. Jedoch werden unter Umständen Zeichen am rechten Ende der Zeile herausgeschoben.

)) n (Push right)
Dieses *line-command* verschiebt den Inhalt eines Blockes von Zeilen zwischen den aktuellen Textgrenzen um n Spalten nach rechts. Dabei gehen im Unterschied zum Kommando > > eventuell Zeichen am rechten Textrand verloren.

(n (Push left)

((n (Push left)
Diese Kommandos arbeiten analog wie) und)) und bewirken eine Verschiebung nach links.

M (Move)
Mit diesem *line-command* wird eine Zeile gekennzeichnet, die an eine andere Position im Text verschoben werden soll.

MM (Move)
Um einen Block zum Verschieben zu kennzeichnen, werden die Kommandos MM in der ersten und letzten Zeile des Blockes eingetragen.

C (Copy)
Hiermit kennzeichnen Sie Zeilen, die Sie an eine andere Position im Text kopieren wollen. Der Unterschied zwischen Kopieren und Verschieben besteht darin, daß beim *line-command* M die ursprüngliche Zeile gelöscht wird, während sie bei C erhalten bleibt (der Text wird also länger).

CC (Copy)
Durch zwei CC-Zeilenkommandos wird ein Block zum Kopieren gekennzeichnet.

Zielposition bei Kopier- und Verschiebebefehlen

Bei den Kopier- und Verschiebebefehlen muß noch eine Zielposition angegeben werden. Dabei gibt es die folgenden drei Zielangaben:

A (After)
Der Block wird hinter diese Zeile kopiert.

B (Before)
Der Block wird vor diese Zeile kopiert.
O (Overlay)
Die Zeile wird *über* diese Zeile kopiert. Das bedeutet, daß Leerzeichen in der mit O gekennzeichneten Zeile durch die Zeichen derselben Spalte in der mit C oder M gekennzeichneten Zeile ersetzt werden.
OO (Overlay)
Der Quellblock (markiert mit M, C, MM oder CC) wird *über* den mit OO markierten Block kopiert, wie dies beim *line-command* O beschrieben wurde. Ist der Quellblock kürzer als der Zielblock, so werden die Zeilen des Quellblockes zyklisch wiederholt.

Reihenfolge der Markierungen

Eine Verschiebe- oder Kopieroperation wird erst dann ausgeführt, wenn sowohl der Zeilenbereich als auch die Zielposition bekannt sind. Die Reihenfolge der Markierung ist unerheblich, Sie können also auch zunächst die Zielposition definieren, und erst anschließend den Quellblock (z.B. mit MM) festlegen. Einige Beispiele sollen die zahlreichen Möglichkeiten zum Verschieben von Textblöcken erläutern.

Beispiele:
Markieren Sie Zeile 5 mit dem line-command C und Zeile 8 mit dem Kommando A. Dadurch wird eine Kopie von Zeile 5 hinter Zeile 8 gestellt.
Markieren Sie Zeile 3 und Zeile 5 jeweils mit dem line-command MM und Zeile 0 mit dem Kommando B. Dadurch werden Zeile 3, 4 und 5 direkt an den Textanfang (vor Zeile 0) kopiert. Außerdem wird der Bereich zwischen Zeile 3 und 5 gelöscht.
Füllen Sie Zeile 0 vollständig mit dem Zeichen »«. Kennzeichnen Sie Zeile 0 mit dem Kommando C. Anschließend markieren Sie Zeile 4 und Zeile 8 jeweils mit dem line-command OO. Nach der Ausführung dieser Kopieroperation ist jedes Leerzeichen in den Zeilen 4 bis 8 durch einen Stern ersetzt worden.*
Nachdem Sie schon einige Zeit mit dem Editor gearbeitet haben, sollten Sie die obigen Beschreibungen noch einmal studieren. Sie werden sicher einige interessante neue Anwendungsmöglichkeiten der Befehle finden, indem Sie z.B. die Textgrenzen in der Statuszeile »BND=« variieren.

4.2.6 Der Textmodus

Um für die verschiedenen Aufgaben bei der Texteditierung eine optimale Cursorsteuerung zu ermöglichen, kennt der Editor zwei verschiedene Eingabemodi.

Eingabemodi des Editors

Beim Start des Editors ist der sogenannte Textmodus aktiviert. In ihm kann man »blind« Texte eingeben, da das Textfenster ständig dem Cursor bei der Eingabe folgt und der Cursor das Textfenster bei der Eingabe nicht verläßt. Außerdem sind die Tabulatoren aktiviert.

Zum Beispiel bei Eingabe von Tabellen ist es jedoch sinnvoller, Textfenster explizit mit den Funktionstasten zu bewegen. Um ein automatisches *scrolling* bei Eingaben im Textfenster zu vermeiden, läßt sich der Textmodus deshalb mit dem *primary-command* RESET ausschalten. Bei Bedarf kann er wieder mit dem *primary-command* TEXT eingeschaltet werden.

»Blinde« Texteingabe im Textmodus möglich

Im Textmodus gelten folgende Regeln für die Texteingabe: Erreicht der Cursor bei der Eingabe von Zeichen im Textfenster den rechten Bildrand, folgt das Textfenster automatisch (um den gewählten Scroll-Betrag) dem Cursor. Beim Erreichen des rechten Textrandes (Markierung > in der Statuszeile »BND=«) springt der Cursor in die erste Textspalte der Folgezeile (Markierung < in der Statuszeile »BND=«). Wird der untere Rand des Bildschirmes erreicht, so wird das Textfenster automatisch um den gewählten Scroll-Betrag weitergeblättert.

Tabulatorsprünge im Textmodus

Im Textmodus springt der Cursor bei der Betätigung der Return-Taste im Textfenster zum Zeilenanfang der Folgezeile. Durch die Eingabe von <Shift> <Return> springt der Cursor auf die nächste vortabulierte Position in der Zeile (Markierung – in der Statuszeile »TAB=«).

4.2.7 FIND

Suche im Quelltext

Mit dem *primary-command* FIND (Kurzform: F) können Sie nach einer Zeichenfolge (*string*) suchen, die maximal 32 Zeichen lang sein darf. Eine Reihe von Parametern können die Suche beeinflussen. Diese Parameter folgen dem Kommando und müssen voneinander durch mindestens ein Leerzeichen getrennt werden. Bei den Parametern ist folgende Reihenfolge einzuhalten:

Reihenfolge der Parameter

```
        String     ALL      CHARS
FIND    'String'   FIRST    WORD     nnnn
         #kkk      NEXT     PREFIX   nnnn-mmmm
          *                SUFFIX
```

Übereinanderstehende Parameter (wie z.B. ALL und NEXT) stellen eine Auswahl dar, von der nur eine Variante pro Befehl angegeben werden darf. Nur der erste Parameter (String, 'String', #kkk bzw. *) muß angegeben werden. Beim Fehlen der übrigen Parameter werden Standardwerte angenommen.

Nachfolgend wird die Bedeutung aller Parameter aufgelistet:

String

Such-Strings

bezeichnet eine Zeichenfolge, deren Länge 32 Zeichen nicht übersteigen darf. Außerdem sind Leerzeichen innerhalb des Strings nicht erlaubt. Die Zeichenfolge darf nicht mit »*«, »#« oder »'« beginnen.

Beispiel:

FIND INTEGER sucht nach der Zeichenfolge INTEGER im Text.

'String'

bezeichnet eine Zeichenfolge, die nicht länger als 32 Zeichen sein darf. Innerhalb der Zeichenfolge sind alle Zeichen außer dem Hochkomma »'« erlaubt.

Beispiel:
FIND 'Ein String mit Leerzeichen' sucht nach der obigen Zeichenfolge. Die Anzahl der Leerzeichen zwischen den Worten muß exakt mit dem Such-String übereinstimmen.

»*«
Der Stern repräsentiert den Such-String des letzten FIND- oder CHANGE-Befehls. Man erspart sich damit die wiederholte Eingabe der Zeichenfolge.

#kkk
Um auch nach Sonderzeichen innerhalb des Textes suchen zu können, die nicht von der Tastatur eingegeben werden können, läßt sich durch diesen Parameter der ASCII-Code eines Zeichens eingeben.

Beispiel:
FIND #27 sucht nach dem Zeichen mit dem ASCII-CODE 27. Dies ist das Zeichen <Esc>, das am Bildschirm als eine invers gedruckte eckige Klammer auf dargestellt wird. Die Codes aller Zeichen des Commodore-Zeichensatzes sind im Anhang A des BASIC-Handbuches aufgelistet.

Suchrichtung

NEXT Kurzform: NEX
Dieser Parameter bewirkt, daß die Suche nach der Zeichenfolge ab der momentanen Cursor-Position beginnt. Befindet sich der Cursor außerhalb des Textfensters, so beginnt die Suche am linken Textrand der ersten Bildschirmzeile. Ohne Angabe einer Startposition (NEXT, FIRST, ALL) wird der Parameter NEXT angenommen.

FIRST Kurzform: FIR
Die Suche beginnt bei der ersten Textspalte der Zeile 0000.

Beispiel:
FIND (FIR sucht nach der ersten öffnenden Kommentarklammer im Text.*

ALL
Es wird gezählt, wie oft der angegebene Such-String im gesamten Text auftritt. Als Ergebnis wird entweder »STRING NOT FOUND« oder »nnnn TIMES FOUND« angezeigt. Der Cursor wird nicht bewegt.

Beispiel:
FIND WRITE ALL zählt, wie oft die Zeichenfolge »WRITE« im gesamten Text auftritt.

Prüfung von Wortgrenzen

CHARS
Der Such-String wird auch gefunden, wenn er Bestandteil eines größeren Wortes ist.

Beispiel:
FIND der CHARS findet auch das Wort »jeder«, da es den String »der« enthält.

WORD Kurzform: WOR
Die Zeichenfolge wird nur gefunden, wenn sie mit einem Trennzeichen beginnt

und endet. Als Trennzeichen gelten alle Zeichen, außer den Buchstaben »A« bis »Z« und »a« bis »z«.

Beispiel:

FIND I WORD findet die Variable »I«, nicht jedoch die Worte »SIN«, »INTEGER« oder »AI«.

PREFIX Kurzform: PRE

Die Zeichenfolge wird nur gefunden, wenn sie mit einem Trennzeichen beginnt, also am Anfang eines Wortes steht.

SUFFIX Kurzform: SUF

Die Zeichenfolge muß mit einem Trennzeichen enden, um gefunden zu werden.

nnnn

Einschränkung des Spaltenbereichs

Wird eine maximal vierstellige Zahl am Ende des Suchkommandos angegeben, so muß der Such-String in der genannten Spalte beginnen. Hat man ein Programm konsequent nach einem festen Schema eingerückt, kann man mit diesem Befehl sehr schnell bestimmte Konstrukte im Quelltext suchen.

Beispiel:

FIND BEGIN 2 sucht das nächste Schlüsselwort BEGIN im Text, das genau zwei Stellen eingerückt ist.

nnnn-mmmm

Um in einem begrenzten Spaltenbereich zu suchen, läßt sich für einen Suchbefehl der Spaltenbereich mit 2 Zahlen einschränken, die ohne Leerzeichen durch einen Bindestrich getrennt sind. Es werden nur Strings gefunden, die mindestens nnnn und maximal mmmm Stellen eingerückt wurden.

Beispiel:

FIND VAR 0-13 sucht nach dem nächsten Schlüsselwort VAR, das maximal 13 Stellen eingerückt ist.

Wird weder der Parameter nnnn noch der Parameter nnnn-mmmm spezifiziert, so werden die Textgrenzen durch die momentan gültigen Textgrenzen aus der Statuszeile nach »BND=« ersetzt.

Beispiel:

FIND PROCEDURE sucht nach dem nächsten Prozedurkopf, der zwischen der Markierung < und der Markierung > in der Statuszeile nach »BND=« steht.

Ergebnis der Suche

Bei der Ausführung des Suchbefehls mit der Tastenkombination <Shift><Return> wird der Cursor folgendermaßen bewegt: Wird bei der Suche vorwärts das Textende (dort steht die Zeile BOTTOM) erreicht, so bleibtder Cursor an seiner Position, und in der Kopfzeile wird die Meldung »** BOTTOM REACHED **« ausgegeben. Begann die Suche bei der ersten Textzeile, wird jedoch die Meldung »STRING NOT FOUND« angezeigt.

Wurde der String jedoch gefunden, so wird der Cursor auf das erste Zeichen des Strings gesetzt und das Textfenster horizontal und vertikal entsprechend

verschoben. In der Kopfzeile erscheint dann die Meldung »STRING FOUND«.
Im Anschluß an ein FIND-Kommando ist die Funktionstaste <F2> aktiviert (siehe Abschnitt 4.2.9).

4.2.8 CHANGE

Suchen und Ersetzen

Das *primary-command* CHANGE erlaubt das automatische Ersetzen von Textteilen durch andere innerhalb des gesamten Textes. Der Aufbau des Befehls und die Regeln für die Reihenfolge der Parameter entsprechen weitgehend dem *primary-command* FIND:

Parameter-Syntax

```
         String1    String2    ALL       CHARS
CHANGE   'String1'  'String2'  FIRST     WORD      nnnn
         #kkk       #kkk       NEXT      PREFIX    nnnn-mmmm
           *          *                  SUFFIX
```

Durchführung der Suche

Es wird zunächst eine Suche nach dem ersten genannten String durchgeführt. Die Parameter (FIRST, NEXT, CHARS, WORD, PREFIX, SUFFIX und nnnn-mmmm) steuern den Ablauf der Suche.
Wird der String nicht gefunden oder das Textende erreicht, erscheinen ebenfalls die Meldungen »STRING NOT FOUND« oder »**BOTTOM REACHED**«. Wird der String1 jedoch gefunden, so wird er durch die zweite Zeichenfolge ersetzt. In diesem Fall erscheint die Meldung »0001 CHANGED«.

Zeilenüberlauf

Ein Sonderfall liegt vor, wenn String2 länger als String1 ist. In diesem Fall wird vor der Umwandlung geprüft, ob am Zeilenende (Markierung > in der Statuszeile nach »BND=«) noch genügend Platz zum Einfügen vorhanden ist. Ist dies nicht der Fall, so wird die Fehlermeldung »MOVE ERROR« angezeigt, ohne daß der String ersetzt wird. Außerdem ist als zweiter String auch ein Leerstring »''« möglich, so daß der Such-String gelöscht wird.
Nach einem CHANGE-Befehl sind die Funktionstasten <F2> und <F4> aktiviert (siehe Abschnitt 4.2.9).
Durch die Angabe des Parameters »ALL« werden alle Strings im gesamten Text umgewandelt. Es erscheint die Meldung »kkkk CHANGED«. Traten bei der Umwandlung Fehler auf, da nicht genügend Platz am Zeilenende vorhanden ist, wird am Ende der Umwandlungen die Meldung »nnnn/mmmm ERRORS« in der Kopfzeile ausgegeben. Das bedeutet, daß nnnn-mal der Such-String gefunden wurde, wobei mmmm Strings *nicht* umgewandelt werden konnten. Oft ist es jedoch sicherer, schrittweise mit den Funktionstasten <F2> und <F4> den Text zu durchlaufen (siehe Beispiel in Abschnitt 4.2.9).

Beispiele:

CHANGE INTEGER REAL FIRST wandelt das erste Wort INTEGER in das Wort REAL um.

CHANGE PROCEDURE FUNCTION ALL wandelt alle Prozeduren des Programmes in Funktionen um.
CHANGE '($R+ *)' " löscht die nächste Zeichenfolge »(*$R+ *)« im Text.*

4.2.9 Die Tasten <F2> und <F4>

Wiederholen der Suche und Ersetzen per Tastendruck

Um die ständige Wiederholung der Eingabe der *primary-commands* FIND und CHANGE zu vermeiden, läßt sich der letzte FIND-Befehl mit der Funktionstaste <F2> und der letzte CHANGE-Befehl mit der Funktionstaste <F4> wiederholen. Dabei werden die Spezifikationen (CHARS/WORD/SUFFIX/PREFIX und die Textspalten nnnn-mmmm) des letzten *primary-commands* FIND bzw. CHANGE verwendet.
Bei der Betätigung der <F2>-Funktionstaste beginnt die Suche ab der momentanen Cursor-Position im Textfenster. Befindet sich der Cursor außerhalb des Textfensters, so beginnt die Suche in der ersten Textzeile, die im Textfenster sichtbar ist.
Wird bei dieser Suche das Textende erreicht, so erscheint die Meldung »**BOTTOM REACHED**«, ohne daß der Cursor bewegt wird. Wird nun die F2-Funktionstaste erneut betätigt, beginnt die Suche in der Textzeile 0000. Sollte der Such-String wiederum nicht gefunden werden, wird in der Kopfzeile die Meldung »STRING NOT FOUND« ausgegeben.
Wie bei den *primary-commands* FIND und CHANGE springt der Cursor bei einer erfolgreichen Suche auf den ersten Buchstaben des Suchwortes, wobei das Textfenster eventuell horizontal oder vertikal verschoben wird. Des weiteren wird in der Kopfzeile »STRING FOUND« angezeigt.
Mit der Funktionstaste <F4> wird ebenfalls eine Suche wie oben beschrieben ausgeführt. Wird jedoch der Such-String im Text gefunden, so wird eine Umwandlung in den String 2 des letzten *primary-commands* CHANGE vorgenommen. Außerdem wird die Meldung »0001 CHANGED« angezeigt. Dabei gelten die in Abschnitt 4.2.8 beschriebenen Regeln für die Umwandlung. Ist insbesondere nicht genügend Platz am Zeilenende, wird ebenfalls keine Umwandlung durchgeführt, sondern die Meldung »MOVE-ERROR« ausgegeben.

Rücksetzen der Tastaturbelegung

Wurde seit dem Aufruf des Editors oder dem letzten RESET-Befehl noch kein FIND- bzw. CHANGE-Befehl verwendet, erscheint bei der Betätigung von <F2> oder <F4> die Fehlermeldung »KEY NOT ACTIVE«.

Beispiel:

```
CHANGE ( [ FIRST
```

Mit diesem primary-command starten Sie die Umwandlung von runden in eckige Klammern bei der Textzeile 0000. Der Cursor steht nach der Ausführung auf der ersten umgewandelten Klammer.

F2-Taste betätigen
Der Cursor springt auf die nächste runde Klammer im Text.
F4-Taste betätigen
Der Cursor bleibt auf der alten Textposition. Es wird die runde in eine eckige Klammer umgewandelt.
F4-Taste betätigen
Der Cursor springt zur nächsten runden Klammer und wandelt sie direkt in eine eckige Klammer um.
F2-Taste betätigen
Der Cursor springt zur nächsten runden Klammer.
F2-Taste betätigen
Die runde Klammer wird nicht umgewandelt. Der Cursor springt zur nächsten runden Klammer.
Dieses Beispiel zeigt, wie man eine Umwandlung eines Zeichens in ein anderes innerhalb des gesamten Textes vornimmt, wobei man an jeder Stelle entscheiden kann, ob das Zeichen wirklich umgewandelt werden soll. Diese Methode ist oft sicherer als der Befehl
`CHANGE ( [ ALL`

4.2.10 INPUT

Einfügen sequentieller Dateien in den Text

Mit dem *primary-command* INPUT (Kurzform: IN) können sequentielle Dateien von beliebigen Peripheriegeräten eingelesen werden. *Vor* der Ausführung des Kommandos können Sie die Einfügeposition mit den *line-commands* A oder B (after und before) festlegen. Beim Fehlen einer solchen Angabe wird der Text der sequentiellen Datei vor Zeile 0000 eingefügt.

Parameter des Files

Nach der Bestätigung des *primary-commands* gelangen Sie in ein Menü, in dem folgende Parameter eingegeben werden, die den Einlesevorgang steuern:

```
DEVICE NUMBER:
['*' OR '?' FOR END]
==>
```

Zunächst bestimmen Sie das Peripheriegerät durch die Angabe seiner Geräteadresse. Die Diskettenstation besitzt z.B. die Geräteadresse 8. Durch Eingabe von »*« oder »?« können Sie den INPUT-Befehl an dieser Stelle abbrechen.

```
SEK. ADDRESS:
['*' OR '?' FOR NONE]
==>
```

Viele Geräte besitzen unter derselben Geräteadresse noch eine sogenannte Sekundäradresse, die an dieser Stelle angegeben werden kann. Wollen Sie keine Sekundäradresse spezifizieren, so können Sie »*« oder »?« eingeben.

```
FILE NAME:
==>
```

Auf Disketten und Kassetten werden Dateien mit einem Namen gespeichert. Beim Laden einer Datei muß der Name an dieser Stelle genannt werden.

```
CODE OF DELIMITER:
['*' OR '?' FOR NONE]
==>13
```

Beschreibung der Satzstruktur durch ein Begrenzungszeichen

Nachdem die Datei vollständig beschrieben wurde, werden Sie jetzt nach dem Code eines Begrenzungszeichens gefragt. Sequentielle Dateien werden meist dadurch in Zeilen (*Sätze* oder *records*) gegliedert, daß am Ende jeder Zeile ein Begrenzungszeichen steht. Üblicherweise handelt es sich bei diesem Zeichen um das sogenannte *carriage return* (Wagenrücklaufcode). Es besitzt den dezimalen ASCII-Code 13. Daher erhalten Sie diesen Wert bei der Eingabe als Vorgabe.

Um jedoch auch Textdateien in exotischen Formaten einlesen zu können, dürfen Sie an dieser Stelle einen beliebigen Code (zwischen 0 und 255) eingeben. Außerdem können Sie mit der Eingabe von »*« oder »?« anzeigen, daß die Datei überhaupt keine Begrenzungszeichen enthält. In diesem Fall werden genau so viele Zeichen in eine Zeile gestellt, wie als maximale Zeilenlänge beim Anlegen des Textes im Editor vereinbart wurde.

Tritt ein Systemfehler beim Einlesen auf, so wird das Einlesen abgebrochen und eine Fehlermeldung in der Kopfzeile angezeigt.

Beispiel:

INPUT mit folgenden Parametern:

```
DEVICE NUMBER:          8
SEK. ADDRESS:           0
FILE NAME:              $0:
CODE OF DELIMITER:      0
```

Damit wird ab der mit dem *line-command* A oder B markierten Zeile das Inhaltsverzeichnis der Diskette im Laufwerk 0 der Diskettenstation mit der Geräteadresse 8 eingelesen. Dieses Verzeichnis ist wie ein BASIC-Programm codiert und enthält daher einige Steuerzeichen, die als invertierte Zeichen dargestellt werden.

Beispiel:

INPUT mit folgenden Parametern:

```
DEVICE NUMBER:          8
SEK. ADDRESS:           3
FILE NAME:              0:ERRORS.TXT,SEQ,READ
CODE OF DELIMITER:      13
```

Es wird die sequentielle Datei »ERRORS.TXT« von der Diskette gelesen. In dieser Datei wird jede Zeile mit einem *carriage-return*, dem Zeichen mit dem ASCII-Code 13, beendet.

Mit dem *primary-command* INPUT können Sie auch Files lesen, die in Pascal-

Programmen mit den Anweisungen WRITE (*File-Variable, Ausdruck*) erzeugt wurden. Dabei erzeugt die Anweisung WRITELN (File-Variable) einen Zeilenwechsel durch die Ausgabe eines *carriage-return*.

4.2.11 OUTPUT

Ausgabe des Textes in eine sequentielle Datei

Mit dem *primary-command* OUTPUT (Kurzform: OUT) wird der Inhalt des Arbeitsspeichers als sequentielle Datei auf einem beliebigen Peripheriegerät gespeichert.

Nach der Bestätigung des *primary-commands* gelangen Sie in ein Menü, in dem folgende Parameter eingegeben werden, die die Ausgabe steuern.

Parameter des Files

```
DEVICE NUMBER:
['*' OR '?' FOR END]
==>
```

Zunächst bestimmen Sie das Peripheriegerät durch die Angabe seiner Geräteadresse. Die Kassette besitzt z.B. die Geräteadresse 1. Durch die Eingabe von »*« oder »?« können Sie den OUTPUT-Befehl an dieser Stelle abbrechen.

```
SEK. ADDRESS:
['*' OR '?' FOR NONE]
==>
```

Mit der Sekundäradresse können Sie z.B. bei Kassettendateien bestimmen, ob eine *end-of-tape*-Markierung geschrieben wird. Wollen Sie keine Sekundäradresse spezifizieren, so können Sie »*« oder »?« eingeben.

```
FILE NAME:
==>
```

Schließlich geben Sie an dieser Stelle an, unter welchem Namen die Datei gespeichert werden soll. Drücken Sie nur die Return-Taste, so wird kein Name vergeben.

```
CODE OF DELIMITER:
['*' OR '?' FOR NONE]
==>13
```

Begrenzungszeichen zur Definition der Zeilenstruktur

Zur Anpassung an die unterschiedlichen Formate bei der Speicherung sequentieller Dateien können Sie jetzt den Code für ein Begrenzungszeichen eingeben, das am Ende jeder Zeile des Textes angefügt werden soll.

Wie bei dem *primary-command* INPUT beschrieben, handelt es sich hierbei üblicherweise um das sogenannte *carriage return* (Wagenrücklaufcode). Es besitzt den dezimalen ASCII-Code 13. Daher erhalten Sie diesen Wert bei der Eingabe als Vorgabe. Durch die Eingabe von »*« oder »?« wird am Ende einer Zeile kein Begrenzungszeichen ausgegeben.

```
TRUNCATE TRAILING
SPACES? [YES OR NO]
==>Y
```

Leerzeichen am Zeilenende

Während die Texte innerhalb des Editors mit einer festen Zeilenlänge gespeichert werden, wird in Textdateien oft auf die Ausgabe von Leerstellen am Zeilenende verzichtet. Zum Beispiel wird die Druckerausgabe durch das Abschneiden nachfolgender Leerzeichen erheblich beschleunigt. Durch die Eingabe des Buchstabens »N« anstelle des vorgegebenen »Y« werden die Zeilen jedoch in voller Länge ausgegeben.

Beispiel:

Die Zeile

»Dies ist eine Zeile mit Leerstellen _____«

wird bei der Eingabe »Y« folgendermaßen ausgegeben:

»Dies ist eine Zeile mit Leerstellen«

Bei der Eingabe »N« werden nachfolgende Leerzeichen nicht abgeschnitten:

»Dies ist eine Zeile mit Leerstellen _____«

Tritt ein Systemfehler bei der Ausgabe auf, so wird die Ausgabe abgebrochen und eine Fehlermeldung ausgegeben.

Beispiel:

SYSTEM ERROR NR:5 zeigt an, daß kein Gerät mit dieser Geräteadresse am Computer angeschlossen ist.

Beispiel:

OUTPUT mit folgenden Parametern:

```
DEVICE NUMBER:                4
SEK. ADDRESS:                 7
FILE NAME:                    *
CODE OF DELIMITER:            13
TRUNCATE TRAILING SPACES:     Y
```

Durch diese Angaben wird der Inhalt des Arbeitsspeichers an den Drucker gesendet. Bei dem Drucker MPS-802 bewirkt die Sekundäradresse 7 eine Ausgabe in Kleinschrift.

Beispiel:

OUTPUT mit folgenden Parametern:

```
DEVICE NUMBER:                8
SEK. ADDRESS:                 3
FILE NAME:                    @0:TEST,SEQ,WRITE
CODE OF DELIMITER:            13
TRUNCATE TRAILING SPACES:     Y
```

Der gesamte Text wird als sequentielles File mit dem Namen »TEST« im Laufwerk 0 auf der Diskettenstation mit der Geräteadresse 8 gespeichert. Eine eventuell zuvor existierende Datei gleichen Namens wird überschrieben.

Mit dem *primary-command* OUTPUT können Sie auch Files erzeugen, die in Pascal-Programmen mit den Anweisungen READ(*File-Variable, Variable*) gelesen werden können.

4.2.12 COPY

Einfügen von Textblöcken von Diskette

Mit diesem *primary-command* können Sie Teile anderer Texte, die auf Diskette gespeichert sind, an einer beliebigen Stelle im Arbeitsspeicher einfügen. Die Einfügeposition wird *vor* der Ausführung des *primary-commands* mit den *line-commands* A oder B definiert. Fehlt eine solche Angabe, wird der Text von Diskette vor der ersten Textzeile (0000) eingefügt.

Editortexte können nicht mit dem Befehl INPUT eingelesen werden, da sie neben dem eigentlichen Text noch Steuerinformationen und den Inhalt der Statuszeilen beinhalten.

In der Kopfzeile können Sie hinter dem COPY-Befehl noch eine ganze Zahl eingeben. Diese legt dann die Geräteadresse der Diskettenstation für den COPY-Befehl fest. Fehlt diese Angabe, so wird der Text vom angemeldeten Laufwerk (siehe Befehl DRIVE in Abschnitt 4.2.4) gelesen.

Wie bei den Befehlen INPUT und OUTPUT gelangen Sie bei der Betätigung mit <Shift><Return> in ein spezielles Menü:

```
COPY DATASET
------------
ENTER DATASET-NAME:
['*' OR '?' FOR END]
==>
```

An dieser Stelle geben Sie den Namen eines Editortextes ein, der auf der eingelegten Diskette im angegebenen Diskettenlaufwerk (mit SAVE) gespeichert wurde. Wird der Text nicht gefunden, so erfolgt ein Rücksprung in das Editorbild. Dies geschieht ebenfalls bei der Eingabe von »*« oder »?«.

Probleme wegen unterschiedlicher Satzlängen

Problematisch ist das Kopieren aus einem Datenbestand, der eine andere Satzlänge als der Text im Arbeitsspeicher besitzt. Hierbei können unter Umständen Zeilen zerstückelt werden. Es erscheint daher zur Warnung folgende Meldung, in der Sie das Kopieren durch die Eingabe des Buchstabens »C« bestätigen müssen:

```
THIS DATASET HAS A
DIFFERENT RECORD-SIZE!
CONFIRM COPY WITH 'C'!
==>
```

Kopieren kleiner Zeilenbereiche

Jede andere Eingabe bricht das Kommando COPY ab. Anschließend können Sie noch den Zeilenbereich festlegen, der kopiert werden soll:

```
FIRST LINE COPIED:
['*' OR '?' TO COPY ALL]
==>
```

Hier geben Sie die Zeilennummer der ersten Zeile ein, die von der Diskette gelesen werden soll. Mit der Eingabe »*« oder »?« wird der gesamte Text von der Diskette gelesen, ansonsten können Sie noch die letzte zu kopierende Zeile definieren:

```
LAST LINE COPIED:
==>
```

Ist die letzte Zeilennummer kleiner als die erste Zeilennummer, oder geben Sie als letzte Zeilennummer »*« oder »?« ein, so können Sie die erste Zeilennummer korrigieren.
Es ist nicht möglich, mit COPY aus einem Text auf Diskette die Statuszeilen (wie z.B. die Tabulatorzeile) zu kopieren. Reicht beim Kopieren der verfügbare Speicherplatz im Arbeitsspeicher nicht aus, so wird die Meldung »OUT OF MEMORY« angezeigt und das Einlesen abgebrochen.

4.2.13 Belegung eigener Funktionstasten

Vordefinierte und frei definierbare Funktionstasten

Während die Funktionstasten <F1>, <F2>, <F3>, <F4>, <F5> und <F7> innerhalb des Editors eine vordefinierte Bedeutung besitzen (siehe Abschnitt 4.2.3 und 4.2.9), können folgende Tasten von Ihnen mit eigenen Texten belegt werden:

1. <Shift><Run/Stop>
2. <Help>
3. <F6>
4. <F8>

Definition eines Kommandomakros

Jedoch lassen sich nicht nur Texte, sondern auch lange Kommandosequenzen (Makros) speichern. Um eine dieser Tasten neu zu definieren, betätigen Sie zunächst die Run/Stop-Taste. Es erscheint folgende Meldung in der Kopfzeile:

```
SELECT KEY-NUMBER (1-4)
```

Nun können Sie eine der *Ziffern*-Tasten <1> bis <4> wählen. Die Ziffer entspricht der Nummer der Funktionstaste in der obigen Liste. (Die Tasten sind auf der Tastatur von links nach rechts durchnumeriert).

Beispiel:

Wählen Sie die Ziffer »2« (Sie können auch die Tasten auf der abgesetzten Zehnertastatur verwenden). Dadurch wird die Help-Taste programmiert.

Haben Sie sich jedoch entschlossen, keine Taste neu zu programmieren, können Sie die Neubelegung mit der Run/Stop-Taste abbrechen. Nachdem Sie eine gültige Ziffer gewählt haben, verschwindet die Auswahlmeldung, und der Inhalt der Kopfzeile wird wiederhergestellt.
Nun befinden Sie sich im Programmiermodus für die gewählte Taste. Sie können jetzt normal mit dem Editor weiterarbeiten, jedoch werden alle nachfolgenden Eingaben von der Tastatur innerhalb des Editors gespeichert.
Diesen Programmiermodus verlassen Sie durch die erneute Betätigung der

Run/Stop-Taste. Jetzt werden alle zwischenzeitlichen Tastatureingaben der entsprechenden Taste (im Beispiel der Taste <Help>) zugewiesen.
Solange Sie vom Editor nicht zum BASIC-Interpreter wechseln, bleibt diese Tastaturbelegung erhalten. Jede Betätigung der Help-Taste wiederholt die gespeicherten Tastatureingaben. Besonders interessant ist die Möglichkeit, während der Programmierung Kommandos und andere Funktionstasten zu speichern. Dadurch lassen sich häufig wiederholte Aktionen zu einem Tastendruck zusammenfassen.
Ein einzelner Tastatur-String darf nicht länger als 64 Zeichen werden. Für alle 4 Funktionstasten dürfen maximal 250 Zeichen definiert werden.

Beispiel:

Sie wollen die Taste <F6> mit dem Text »HALLO OTTO« belegen, da Sie diesen Text mehrmals im Textfenster eingeben wollen.
Zunächst gehen Sie an die Textposition, an welcher der Text zum ersten Mal verwendet werden soll. Die Programmierung der Taste starten Sie mit der Taste <Run/Stop>:

```
SELCET KEY-NUMBER (1-4)
```

Jetzt drücken Sie die Zifferntaste <3>, um die nachfolgenden Tastatureingaben auf der dritten frei belegbaren Funktionstaste <F6> zu speichern. Die angezeigte Meldung wird wieder ausgeblendet. Anschließend geben Sie den Text »HALLO OTTO« ein.
Um die Programmierung zu beenden, drücken Sie noch einmal auf die Run/Stop-Taste. Fortan müssen Sie statt »HALLO OTTO« einzugeben nur noch die Taste <F6> betätigen. Den Text können Sie überall (auch in der Kopfzeile) verwenden.

Beispiel:

Sie wollen mit der Funktionstaste <F8> zum Textanfang blättern. Nachdem Sie mit der Run/Stop-Taste und der Zifferntaste <4> die Programmierung eingeleitet haben, bewegen Sie den Cursor mit der Taste <Shift><CLR> in die Kopfzeile und geben dort den Text »LOCATE 0« ein. Führen Sie das Kommando mit <Shift><Return> aus. Jetzt beenden Sie noch die Programmierung mit der Run/Stop-Taste. Von nun an können Sie an jeder Bildschirmposition mit der Betätigung der F8-Taste zum Textanfang blättern.

Schnelle Cursorbewegung

Eine nützliche Tastenbelegung besteht darin, die Taste <F6> mit 10 Cursorbewegungen nach links und die Taste <F8> mit 10 Cursorbewegungen nach rechts zu belegen. Damit kann man sich im 80-Zeichen-Modus sehr schnell horizontal im Text vorwärts und rückwärts bewegen. Wenn Sie längere Zeit mit dem Editor gearbeitet haben, werden Sie sicher oft von den Möglichkeiten der definierbaren Funktionstasten Gebrauch machen.

4.2.14 Meldungen innerhalb des Editors

Folgende Meldungen können im Editor auftreten. Zu jeder Meldung werden ein kurzer Hinweis zu ihrer Bedeutung und eventuelle Korrekturmöglichkeiten gegeben. Nähere Informationen finden Sie bei der Beschreibung der Kommandos, bei deren Ausführung der Fehler gemeldet wurde.

Meldung	**Erklärung und Korrekturmöglichkeiten**
LINE-COMM. IGNORED	Es wurden zu viele *line-commands* eines Typs angegeben. Die erkannten *line-commands* werden nicht ausgeführt, sondern nochmals angezeigt. Diese Meldung tritt auch auf, falls Sie z.B. zu einem DD-Kommando statt des zugehörigen DD- ein D-Kommando eingeben. Um unvollständige *line-commands* zu löschen, können Sie das *primary-command* RESET verwenden.
COMMAND CONFLICT	Bei den *line-commands* MM und CC liegt die A-oder B-Markierung innerhalb des Quellblockes.
OUT OF MEMORY	Der Arbeitsspeicher ist zu klein, um den Text mit den *line-commands* I, R oder C zu erweitern. Bei dem *line-command* MM hilft in diesem Fall ein Zerlegen des Blockes in kleinere Teile.
BLOCK TOO LONG	Ein mit den *line-commands* MM, CC, >>, RR etc. definierter Block darf nicht mehr als 256 Zeilen umfassen. Sie müssen die Operation mit mehreren Blockbefehlen durchführen.
ILLEGAL BOUNDS	In der Statuszeile nach »BND=« wurden ungültige Eingaben gemacht (siehe 4.2.2.3).
MOVE ERROR	Bei den *line-commands* > oder < würden Zeichen über den Textrand verschoben. Dieser Fehler tritt auch bei dem *primary-command* CHANGE auf, falls in der Einfügezeile nicht genügend Platz am Zeilenende vorhanden ist.
ILLEGAL COMMAND	Bei den *primary-commands* FIND und CHANGE wurden ungültige Parameter angegeben.
ENTER A STRING!	Ein String bei den *primary-commands* FIND oder CHANGE hat nicht die korrekte Form (es fehlt z.B. ein Apostroph »'«).
ILLEGAL COLUMN	Eine Spaltenangabe bei FIND und CHANGE (nnnn-mmmm) liegt außerhalb des Bereiches 0 und der maximalen Zeilenlänge.
STRING FOUND	Meldung bei dem *primary-command* FIND. Der Cursor steht auf dem gesuchten String im Textfenster.
** BOTTOM REACHED **	Bei FIND oder CHANGE wurde das Textende (dort steht die Zeile BOTTOM) erreicht.
STRING NOT FOUND	Der Such-String befindet sich nicht im Bereich, der durch die Textgrenzen in der Statuszeile »BND=« festgelegt wird.
KEY IS NOT ACTIVE	Die Funktionstasten <F2> und <F4> sind nur aktiviert, wenn zuvor das *primary-command* FIND oder CHANGE verwendet wurde.
nnnn TIMES FOUND	Diese Meldung wird nach dem Kommando FIND ... ALL ausgegeben. nnnn gibt an, wie oft der Such-String gefunden wurde.
nnnn CHANGED	nnnn gibt die Anzahl der geänderten Strings bei einem CHANGE-Befehl an.

nnnn/mmmm ERRORS	Bei dem *primary-command* CHANGE. ALL wurden nnnn Strings gefunden, von denen mmmm *nicht* geändert werden konnten.
SYSTEM ERROR NR:n	Bei einem Betriebssystem-Aufruf (bei den *primary-commands* INPUT und OUTPUT) trat ein Fehler auf. Die Fehlerursache hängt von dem angesprochenen Peripheriegerät ab, so daß keine allgemeingültigen Aussagen zur Fehlerbeseitigung gemacht werden können.
	n Fehlermeldung 1 TOO MANY FILES OPEN 2 FILE OPEN 3 FILE NOT OPEN 4 FILE NOT FOUND 5 DEVICE NOT PRESENT 6 NOT INPUT FILE 7 NOT OUTPUT FILE 8 MISSING FILE-NAME 9 ILLEGAL DEVICE-NUMBER
SELECT KEY-NUMBER (1–4)	Diese Meldung wird nach der Betätigung der Run/Stop-Taste im Editor angezeigt. Durch die Eingabe einer Zahl zwischen 1 und 4 können Sie eine Funktionstaste belegen (siehe Abschnitt 4.2.13). Um die Belegung abzubrechen, genügt eine nochmalige Betätigung von <Run/Stop>.

Außerdem werden bei Diskettenoperationen noch Meldungen der Floppy im folgenden Format angezeigt (nähere Erläuterungen finden sich im Handbuch der Diskettenstation):

`Fehlernummer, Fehlertext, Spur, Sektor`

4.3 Bedienung des Compilers Pascal 2.0

Aufgaben des Compilers

Der Compiler übersetzt den Quelltext, der sich im Arbeitsspeicher befindet, in einem Durchlauf (*one pass*) in ein direkt ausführbares Programm, das wie ein BASIC-Programm gestartet und gespeichert werden kann. Quelltext, Objektprogramm, Editor und Compiler residieren gleichzeitig im Arbeitsspeicher, so daß während der Programmentwicklung auf Diskettenzugriffe verzichtet werden kann.

4.3.1 Wahl der Optionen

Der Compiler wird aus dem Pascal-Menü (siehe Abschnitt 4.2.1) mit der Eingabe »$« (beendet mit der Return-Taste) aufgerufen. Alle Ausgaben des Compilers erfolgen in dem Bildschirmmodus, in dem der Compiler aufgerufen wurde. War zum Beispiel im Pascal-Menü der 40-Zeichen-Bildschirm aktiviert, so erscheinen die im Folgenden beschriebenen Dialoge auf diesem Bildschirm.

Verdoppelung der Übersetzungsgeschwindigkeit im 80-Zeilen-Modus

Sollten Sie über einen 80-Zeichen-Monitor verfügen, ist es sinnvoll, den Rechner mit einer Taktfrequenz von 2 MHz (Kommando FAST in BASIC oder Pascal) zu betreiben, da dadurch die Compilergeschwindigkeit praktisch verdoppelt wird.

Beim ersten Start des Systems nach dem Laden von der Diskette erscheint ein Copyright-Hinweis. Der Compiler meldet sich mit der folgenden Meldung:

```
PASCAL 2.0 (Rx.x)
----------------

SELECT OPTION:
0      CHECK SYNTAX
1      GENERATE CODE
ELSE   LOCATE ADDRESS

==>1
```

Wahl des Übersetzungsmodus

»x.x« bezeichnet dabei den *release* des Programmes. Durch die Eingabe einer ganzen Zahl können Sie den Übersetzungsmodus des Compilers wählen. Innerhalb des Compilers müssen *alle* Eingaben mit der Return-Taste abgeschlossen werden.

Syntaxüberprüfung

Durch die Eingabe der Zahl 0 wird der Quelltext im Arbeitsspeicher mit eventuell vorhandenen INCLUDE-Files gelesen und auf syntaktische Korrektheit geprüft. Alle Fehler werden im Quelltext markiert. Jedoch wird *kein* ausführbarer Code erzeugt. Diesen Modus können Sie verwenden, um ein Programm auf Korrektheit zu überprüfen, ohne ein bereits zuvor erzeugtes Objektprogramm im Arbeitsspeicher zu überschreiben. Der Geschwindigkeitsgewinn gegenüber einer Übersetzung mit Codeerzeugung ist nur minimal.

Codeerzeugung

Mit der Zahl 1 wählen Sie den voreingestellten Compilermodus: Es wird der Quelltext gelesen, auf syntaktische Korrektheit geprüft, und auftretende Fehler werden im ausgegebenen Programmlisting markiert. Werden keine Fehler entdeckt, so steht am Ende der Übersetzung ein ausführbares Objektprogramm im Arbeitsspeicher.

Suche nach Laufzeitfehlern

Jede andere Zahl als 0 oder 1 wird als eine Fehleradresse im Objektprogramm interpretiert, deren Lage im Quelltext lokalisiert werden soll. Bei dieser Eingabe liest der Compiler ebenfalls den Quelltext, ohne jedoch Code zu erzeugen. Statt dessen wird die Position des Laufzeitfehlers bestimmt. Diese Fehlerposition wird mit der Fehlernummer 0 und dem Fehlertext »ADDRESS OF RUNTIME ERROR« markiert. Beachten Sie bitte, daß eine Fehleradresse auch negativ sein kann, da die Adresse im Zweierkomplement dargestellt wird.

Nähere Informationen zum Thema Laufzeitfehler und Beispiele finden Sie in den Abschnitten 1.3.4 und 4.3.3.

Ausgabe eines Programmlistings

Nach der Wahl des Compilermodus können Sie noch wählen, ob das Programmlisting auf dem Drucker ausgegeben werden soll:

```
LISTING TO PRINTER?
==>N
```

Durch die Eingabe eines anderen Buchstabens als »N« an dieser Stelle wird der Quelltext einschließlich INCLUDE-Files mit eventuell vorhandenen Fehlermeldungen und einer Übersicht über die Größe des erzeugten Codes auf das Peripheriegerät mit der Geräteadresse 4 an der seriellen Schnittstelle (das ist normalerweise der Drucker) ausgegeben.

Bestätigen Sie nach dem Aufruf des Compilers die beiden Vorgaben (»1« und »N«) einfach durch Betätigung der Return-Taste, so wird der Quelltext in ein Objektprogramm im Arbeitsspeicher übersetzt, wobei gleichzeitig ein Programmlisting mit Meldungen des Compilers am Bildschirm ausgegeben wird.

4.3.2 Meldungen innerhalb des Compilers

Während der Kompilation wird der Quelltext zeilenweise am Bildschirm angezeigt. Bei der Umschaltung auf INCLUDE-Files (siehe Abschnitt 4.4.6.2) wird auch der von der Diskette gelesene Text angezeigt.

Behandlung von Syntaxfehlern

Bei der Übersetzung jedes Programmblockes wird einmal geprüft, ob die Run/Stop-Taste betätigt wurde. In diesem Fall wird folgende Bestätigungsmeldung ausgegeben:

```
BREAK IN xxxx
'*' STOP
ELSE CONTINUE
==>
```

Wollen Sie die Übersetzung tatsächlich abbrechen, so geben Sie jetzt »*« ein. Bei jeder anderen Eingabe setzt der Compiler seine Arbeit fort. Wird die Übersetzung abgebrochen, so kehren Sie direkt zum Pascal-Menü zurück. In diesem Fall steht natürlich kein ausführbares Objektprogramm im Arbeitsspeicher.

Stellt der Compiler einen Fehler im Quelltext fest, so wird mit einem Pfeil unterhalb der Zeile das Symbol markiert, bei dessen Verarbeitung der Fehler entdeckt wurde:

Beispiel:

In dieser Anweisungsfolge fehlt zwischen den Zuweisungen ein Semikolon:

```
BEGIN X:=Y  Y:=Z END;
            ↑ ERROR 14 IN 13
```

Fehlernummern

Die Fehlernummern werden im Anhang erläutert. Ein generelles Problem der

Sprache Pascal besteht darin, daß Fehler in der Blockschachtelung unter Umständen erst lange nach der eigentlichen Fehlerursache vom Compiler entdeckt werden können. Sollten Sie z.B. die folgende Fehlermeldung erhalten,

```
END;
   ↑ ERROR 20 IN 99
```

so finden Sie als Erklärung für die Fehlernummer 20 im Anhang B den Text »'.' erwartet«. Mit großer Wahrscheinlichkeit bestand aber der Fehler darin, daß Sie einige Zeilen zuvor das Wortsymbol END überflüssigerweise angegeben hatten, so daß der Compiler an dieser Stelle den Punkt am Programmende erwartet. Deshalb sollten Sie Ihre Fehlersuche nicht ausschließlich auf die mit einem Pfeil markierte Textposition beschränken.

Zeilennummern bei Fehler

Neben der Fehlernummer wird noch eine Zeilennummer angegeben. Sie bezieht sich immer auf die innerhalb des Editors angezeigten Zeilennummern, die von 0 in Einerschritten gezählt werden. Fehler in einem INCLUDE-File auf der Diskette werden mit einer Zeilennummer relativ zum Anfang des INCLUDE-Files markiert, so daß Sie die Zeilennummer für die Korrektur des Files im Editor kennen.

Nach dem Auftreten eines Fehlers besitzen Sie folgende Wahlmöglichkeiten:

```
'*' STOP  '?' EXPLANATION
ELSE CONTINUE
==>
```

Fehlermeldung im Klartext

Mit der Eingabe eines Sterns (»*«) brechen Sie die Übersetzung ab und kehren direkt zum Pascal-Menü zurück. Zu jeder Fehlernummer können Sie durch die Eingabe eines Fragezeichens (»?«) einen kurzen erläuternden Text erhalten. Diese Texte sind in englischer Sprache in dem File »ERRORS.TXT« auf Diskette gespeichert. Ist dieses File nicht auf der eingelegten Diskette vorhanden, wird statt einer Erläuterung der Text

```
NO MESSAGE FOUND
```

angezeigt. Geben Sie weder »*« noch »?« ein, so wird die Übersetzung fortgesetzt. Gelegentlich können auf einen Fehler mehrere Folgefehlermeldungen auftreten, die Sie durch einfache Betätigung der Return-Taste übergehen können.

Nach dem Auftreten eines Fehlers prüft der Compiler weiterhin nur noch auf syntaktische Korrektheit, ohne Code zu erzeugen.

Position eines Laufzeitfehlers

Wurde die Adresse eines Laufzeitfehlers gefunden (Modus LOCATE ADDRESS), so erscheint die Meldung:

```
WRITELN(1/0);
      ↑ ERROR 0 in xxxx
```

Zur Fehlernummer 0 erhalten Sie folgende Erläuterung

```
ADDRESS OF RUNTIME ERROR FOUND
```

Anschließend kehren Sie direkt zum Pascal-Menü zurück.

Codegröße

Wurden keine Fehler im Programm gefunden, so steht am Ende der Übersetzung ein ablauffähiges Programm im BASIC-Arbeitsspeicher. Dieses Programm besteht aus einem kleinen Vorspann und dem eigentlichen Pascal-Code. Einen Teil des Codes bilden kompilierte Konstanten, die beim Programmstart in Bank 1 kopiert werden (siehe Abschnitt 4.3.4 über die Speicherverteilung bei Pascal-Programmen). Die Lage und Größe des erzeugten Code können Sie der folgenden Meldung entnehmen:

```
ERRORS DETECTED: 0
     P-CODE FROM 7391 TO xxxx
  CONSTANTS FROM 8192 TO yyyy

(HIT RETURN FOR MENU)
```

Die Rückkehr zum Pascal-Menü erfolgt erst, wenn Sie die Return-Taste betätigt haben.

Temporäre Dateien bei langen Programmen

Wenn Sie Programme entwickeln, die länger als 25 Kbyte werden, reicht der erzeugte Programmcode in Bank 0 bis in den für Quelltexte reservierten Speicherbereich hinein (siehe Abschnitt 4.3.4). In diesem Fall beginnt der Compiler, zwei temporäre Dateien mit den Namen »C$« und »F$« auf der eingelegten Diskette zu erzeugen. Nachdem das Programm fehlerfrei übersetzt wurde, wird am Ende der Kompilation der Inhalt der temporären Dateien geladen. Dadurch wird der Quelltext des Programmes überschrieben und die Auswahlmöglichkeit »?« (RESUME EDIT) im Pascal-Menü gesperrt. Zur Warnung erscheint die Meldung:

```
SOURCE OVERWRITTEN
```

Sollten Sie Programme größer als 48 Kbyte entwickeln, so wird beim Nachladen der temporären Dateien »C$« und »F$« auch der Editor überschrieben. Deshalb wird zunächst die Warnmeldung

```
EDITOR OVERWRITTEN
```

ausgegeben. Bei der Rückkehr vom Compiler wird das Pascal-System automatisch ausgeschaltet. Nach der Copyrightmeldung des BASIC-Interpreters erscheint die Meldung:

```
SYSTEM SWITCHED OFF
```

Das gesamte Objektprogramm steht dann, wie im folgenden Kapitel beschrieben, zum Ausführen und Speichern im Arbeitsspeicher. Vor einer erneuten Übersetzung muß jedoch das Pascal-System wieder von Diskette geladen werden.

4.3.3 Rückkehr zu BASIC

Speicherorganisation in BASIC

Bei der Eingabe »*« im Pascal-Menü verlassen Sie das Pascal-System. Alle nachfolgenden Eingaben werden vom BASIC-Interpreter ausgewertet. Es erscheint die übliche BASIC-Einschaltmeldung:

```
COMMODORE BASIC V7.0 122365 BYTES FREE
  (C)1985 COMMODORE ELECTRONICS, LTD.
        (C)1977 MICROSOFT CORP.
          ALL RIGHTS RESERVED
```

Der Rechner befindet sich auch in demselben Zustand wie direkt nach dem Einschalten, jedoch mit folgenden Unterschieden: In der Speicherbank 0 ist Speicher für den Editor reserviert worden, während in Speicherbank 1 der Compiler und das File »PAS.LIB« gespeichert sind.

Zur Ausführung des übersetzten Programmes genügt die Eingabe des BASIC-Befehls RUN, da der Objektcode einen Vorspann besitzt, der unter anderem aus einer BASIC-Zeile besteht. Diese können Sie sich mit dem BASIC-Befehl LIST anzeigen lassen:

```
1986 GRAPHICCLR:BANK0:SYS7200: PASCAL 2.0
```

Dieses Programm sollten Sie nicht verändern oder löschen, da dieser Vorspann für spätere Übersetzungen wiederverwendet wird. Übersetzte Pascal-Programme können auch wie BASIC-Programme gespeichert und geladen werden:

Speichern des Objekt-programms

```
DSAVE"TEST.M"
DLOAD"TEST.M"
```

Damit Sie später den Quelltext und das Objektprogramm leicht identifizieren können, bietet sich folgende Namenskonvention an: Beim Erstellen von Quelltexten mit dem Editor verwenden Sie das Suffix ».P«, während Sie Objektprogramme mit dem Suffix ».M« speichern.

Start des Objekt-programms

Beim Start eines Objektprogrammes wird zunächst geprüft, ob das File »PAS.LIB« bereits geladen wurde. Bei Bedarf wird es von der Diskette nachgeladen. Daher muß diese Datei auf jeder Diskette, die Pascal-Programme enthält, ebenfalls vorhanden sein. Beim Start des Pascal-Systems wird die Datei »PAS.LIB« automatisch geladen.

Wird diese Datei auf der Diskette nicht gefunden, erscheint die Meldung

```
PAS.LIB NOT FOUND ON DISK
```

In diesem Fall können Sie eine Diskette mit dem File »PAS.LIB« einlegen und das Programm erneut mit RUN starten.

Rückkehr ins Pascal-Menü

Um von BASIC (nach der Meldung READY.) wieder in das Pascal-Menü zu gelangen, genügt die Eingabe eines Sterns »*« am Anfang einer Zeile, die mit <Return> beendet wird.

Ausschalten des Systems

Möchten Sie das Pascal-System ganz ausschalten, so geben Sie zwei Sterne »**« am Anfang der Zeile an. Dadurch erzeugen Sie denselben Zustand wie nach dem Einschalten des Rechners, jedoch bleibt ein eventuell vorhandenes Objektprogramm im Speicher erhalten. Das Pascal-System verabschiedet sich mit der Meldung:

```
COMMODORE BASIC V7.0 122365 BYTES FREE
  (C)1985 COMMODORE ELECTRONICS, LTD.
        (C)1977 MICROSOFT CORP.
          ALL RIGHTS RESERVED
SYSTEM SWITCHED OFF
READY.
```

Treten Fehler während der Ausführung des Objektprogrammes auf, so erscheint eine der in Anhang C aufgeführten Laufzeitfehlermeldungen. Neben der Adresse des Fehlers im übersetzten Code wird auch eine Auflistung der Aufrufadressen (*dynamic chain*) ausgegeben:

Beispiel:

```
PROGRAM LAUFZEITFEHLER(INPUT,OUTPUT);
FUNCTION KEHRWERT(X: INTEGER): REAL;
BEGIN
  KEHRWERT:= 1 / X;
END; (* KEHRWERT *)
PROCEDURE P;
BEGIN
  WRITELN(KEHRWERT(0));
END; (* P *)
BEGIN
 P;
END.
```

Dieses Programm können Sie ohne Fehlermeldung übersetzen lassen. Der erzeugte Code liegt im Bereich von 7391 bis 7473:

```
ERRORS DETECTED: 0
     P-CODE FROM  7391 TO 7473
  CONSTANTS FROM  8192 TO 8192
```

Fehlermeldung bei Laufzeitfehlern

Bei der Ausführung des Programmes werden Sie jedoch feststellen, daß der Prozeduraufruf P im Hauptprogramm eine Division durch null in der Funktion KEHRWERT zur Folge hat. Daher erscheint folgende Meldung nach dem Programmstart mit RUN:

```
DIVISION BY ZERO
 ERROR AT 7437 IN KEHRWERT
CALLED AT 7453 IN P
CALLED AT 7469 IN LAUFZEITFEHLER
READY.
```

Suche nach einer Fehleradresse

Natürlich läßt sich bei diesem kleinen Programm der Fehler innerhalb der Funktion KEHRWERT sofort lokalisieren, jedoch kann es notwendig sein, die exakte Fehlerposition zu finden. Deshalb können Sie den Compiler vom Pascal-Menü starten und die Fehleradresse 7437 eingeben:

```
PASCAL 2.0 (Rx.x)
-----------------
SELECT OPTION:
0       CHECK SYNTAX
1       GENERATE CODE
ELSE    LOCATE ADDRESS

==>7437
```

Die Fehleradresse wird dann mit einem Pfeil und der Fehlernummer 0 vom Compiler im Quelltext markiert:

```
KEHRWERT:= 1 / X;
                 ↑ ERROR 0 IN 4
```

Mit der eben beschriebenen Methode lassen sich **alle** beim Programmabbruch genannten Adressen (also auch die Aufrufadressen der Prozeduren) lokalisieren.

Abbruch der Programmausführung

Laufende Pascal-Programme können nur mit der Tastenkombination <Run/Stop> und <Restore> abgebrochen werden. Auch in diesem Fall wird die Adresse des zuletzt ausgeführten Befehls und die *dynamic-chain* angezeigt.

```
BREAK
 ERROR AT 7468 IN TESTPROGRAMM
```

4.3.4 Speicherverteilung im Pascal-System

Möchten Sie reine Pascal-Programme mit dem Compiler entwickeln, so können Sie das folgende Kapitel überschlagen, da es Hinweise enthält, die nur für denjenigen gedacht sind, der Pascal-Programme mit Programmen in Maschinensprache koppeln möchte. Nur dann ist es nämlich wichtig, die Speicherverwaltung bei übersetzten Pascal-Programmen zu kennen.

Speicherverteilung ohne Pascal-System

Zunächst soll die Speicherverteilung während der Ausführung eines übersetzten Programmes beschrieben werden, ohne daß das Pascal-System vorhanden ist. Von der Diskette laden Sie eine Mischung aus BASIC-Programm, Maschinenprogramm, Pascal-Code und kompilierten Konstanten. Dieser Code *muß* ab der absoluten Adresse $1C01 in der Bank 0 geladen werden. Das Ende des Codes erfahren Sie bei der letzten Meldung des Compilers.

Beim Programmstart wird zunächst die Bibliothek »PAS.LIB« von der Diskette in Bank 1 ab der absoluten Adresse $0400 geladen. Dieses File enthält die gesamte Pascal-Laufzeitbibliothek. Anschließend werden *alle* Konstanten in Bank 1 (hinter PAS.LIB) kopiert. Den Anfang und das Ende der Konstanten in Bank 1 erfahren Sie ebenfalls in der letzten Meldung des Compilers.

heap und stack

Während der Ausführung des Pascal-Codes werden zwei Speicherbereiche in Bank 1 verwaltet. Auf dem *stack* werden alle lokalen Variablen und Zwischenergebnisse gespeichert. Der *stack* beginnt direkt hinter den Konstanten und wächst von niedrigen zu hohen Speicheradressen. Dynamische Variablen (erzeugt mit der Standardprozedur NEW) werden auf dem *heap* verwaltet. Dieser wächst in Bank 1 vom oberen Speicherende abwärts. Bei der Kollision von *heap* und *stack* wird ein Speicherüberlauf gemeldet. Der Speicherbereich ab der Adresse $FF00 in Bank 0 und Bank 1 ist für das Betriebssystem reserviert. Damit ergibt sich die in Bild 115 gezeigte Speicherverteilung:

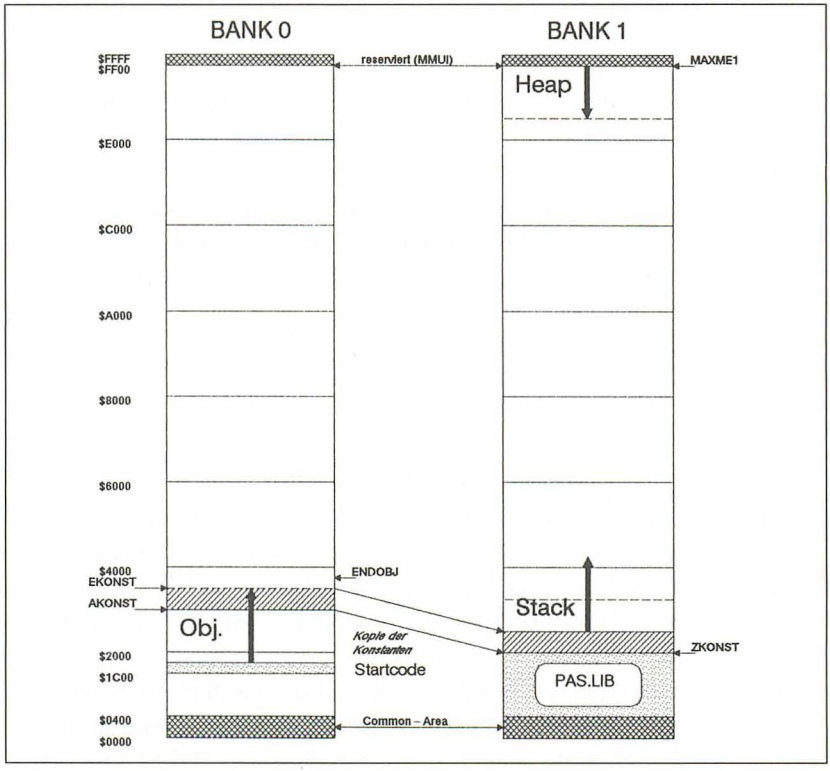


Bild 115: Speicherverteilung ohne Pascal-System

Speicherverteilung bei Grafikprogrammen

Bei Programmen, die Darstellungen im Grafikmodus verwenden, wird in Bank 0 Speicherplatz am Beginn des BASIC-Arbeitsspeichers benötigt. Solche Programme müssen deshalb im Programmkopf die Kennzeichnung GRAPHIC erhalten. Sofort beim Programmstart (nach der Kopie der Konstanten) wird dann das Pascal-Programm in BANK 0 um 8 Kbyte nach oben geschoben. Jedoch ist der Pascal-Code *nicht* relokatibel.

Speicherverteilung im Pascal-System

Ist bei der Ausführung des Objektcodes noch das Pascal-System geladen, so endet der verfügbare Arbeitsspeicher in Bank 1 bei der Adresse $9B00, da dort der Compiler beginnt. Andererseits muß in Bank 0 noch der Source-Text und

der Editor untergebracht werden. Der Source-Text beginnt bei der absoluten Adresse $8000. Der Editor selbst beginnt bei der Adresse $DB00. Schließlich wird der Speicherbereich zwischen $1400 und $1600 in Bank 0 für das Pascal-System reserviert. Summa summarum ergibt sich die in Bild 116 gezeigte *memory-map*:

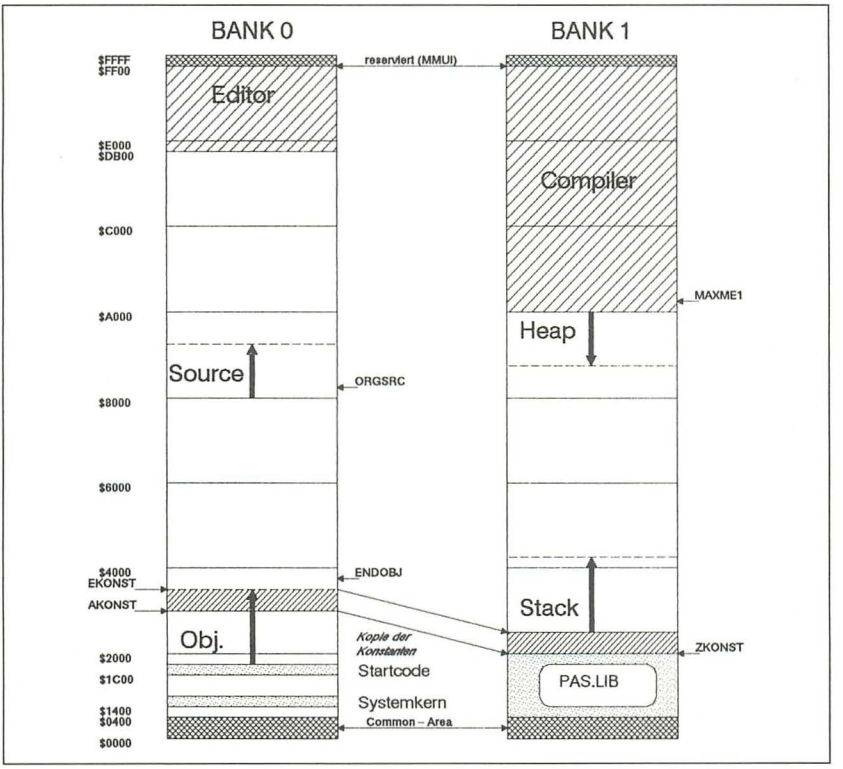


Bild 116: Speicherverteilung im Pascal-System

Es folgt eine Auflistung aller überhaupt vom Laufzeitsystem verwendeten (Zero-Page-)Variablen. Dabei sind nur die Zeiger T, P, B, B1, F, MAXME0, MAXME1 und TOP *innerhalb von SYS-Befehlen* vor Veränderungen zu schützen.

Variablen des Laufzeitsystems

Name	Adresse	Bedeutung
T	47/48	top of stack (Bank 1 aufwärts wachsend)
P	49/50	program counter (Bank 0)
B	59/60	base register (Bank 1)
B1	51/52	help pointer to stack (Bank 1)

Name	Adresse	Bedeutung
F	55/56	file descriptor (Bank 1)
MAXME0	4626/4627	top of BASIC ram (Bank 0)
MAXME1	57/58	top of free ram (Bank 1)
TOP	53/54	top of heap (Bank 1 abwärts wachsend)
VAR	67/77	variable assignation
FAC	99..104	floating point accu
ARG	106..111	floating point accu (2)
SIGN	112	SGN(FAC * ARG)
FACLSB	113	LSB of FAC
TAB	256	scratch area
TABI	512	scratch area for input
FARCNF	2	
FARPCH	3	
FARPCL	4	für bankübergreifende Sprünge
FARST	5	
IODEV	21	
DIRECT	127	
M4080	216	
HIRES	215	
MMUCR	65280..65284	

Alle Adressen sind dezimal angegeben. 2-Byte-Werte sind in der Reihenfolge L/H gespeichert.

Platz für Maschinenprogramme

Um Speicherplatz für Unterprogramme in Maschinenprogrammen zu reservieren, bietet sich das obere Speicherende in Bank 1 und Bank 0 an. Sie müssen dazu nur vor der ersten NEW-Operation die Zeiger MAXME0 bzw. MAXME1 modifizieren.

Jedoch können der Pascal-Code, die Konstanten und der Stack bei Bedarf auch an einer anderen Stelle gespeichert werden. Diese Modifikationen werden durch Parameter im Programmkopf gesteuert. Neben den Parametern INPUT, OUTPUT und GRAPHIC sind in *Pascal 2.0* in der Parameterliste des Programmes ganze Zahlen zulässig, die getrennt durch Semikola die Codeerzeugung beeinflussen.

Beispiel:

Modifikation der Speicherverteilung

```
PROGRAM MODIFIKATION(INPUT,OUTPUT; 10000; 24576; 1000);
```

Die erste Adresse gibt an, ab welcher absoluten Adresse in Bank 0 der Code bei seiner Ausführung stehen soll. Der zweite Parameter gibt an, wohin das erste Byte der Konstanten in Bank 1 kopiert werden soll. Schließlich definiert der dritte Parameter, um welches Offset der Code in Bank 0 gegenüber der ersten Zahl verschoben gespeichert werden soll.

Im Beispiel wird der Compiler den erzeugten Code ab der Adresse 11 000 (= 10 000 + 1000) speichern. Die Konstanten beginnen in Bank 1 ab der Adresse $6000 (=24 576). *Jedoch ist der Code nur lauffähig, wenn er vor der Ausführung wieder an die Adresse 10 000 verschoben wird.*
Nach der Übersetzung muß noch der Vorspann, in dem sich der Code zum Laden des Files »PAS.LIB« befindet, angepaßt werden, damit der Pascal-Code im Speicher gefunden wird. Daher müssen Sie ab der Adresse $1CDF (dies ist der *normale* Codeanfang) mit dem ROM-Monitor folgende Bytes definieren:

$1CDF	AKONST	Anfang der Konstanten in Bank 0 (L/H)
$1CE1	EKONST	Ende der Konstanten in Bank 0 (L/H)
$1CE3	ZKONST	Anfang der Kopie der Konstanten in Bank 1 (L/H)
$1CE5	PCODE	Anfang des Programmcodes in Bank 1 (L/H)

Diese 8 Byte werden bereits vom Compiler mit dem Pascal-Code erzeugt, so daß Sie im obigen Beispiel mit folgendem Transferbefehl (im Monitor) diese Bytes kopieren können:

```
T +11000,+11007,$1CDF
```

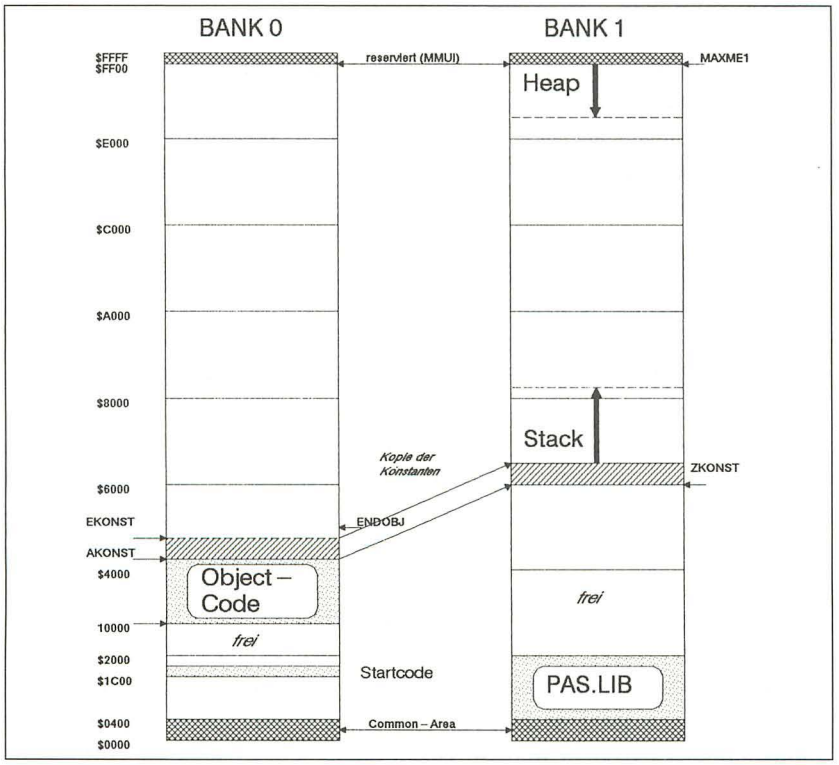


Bild 117: Beispiel einer modifizierten Speicherverteilung

Damit erhalten Sie zur Laufzeit folgende Speicherverteilung, wobei Sie zwischen $1CE7 und +11 000 in Bank 0 sowie $2000 und $6000 in Bank 1 Speicherplatz für eigene Maschinencode-Routinen besitzen.

Zusätzliche Modifikation des Stack

Mit dieser Modifikation beginnt der *stack* direkt hinter der Kopie der Konstanten. Der Stack-Anfang läßt sich jedoch ebenfalls (innerhalb Bank 1) verschieben. Dazu muß in der durch PCODE indizierten Adresse die Basisadresse des *stack* verändert werden. Diese doch recht seltene Modifikation wird in Abschnitt 3.4 in Form von BASIC-Befehlen durchgeführt.

4.4 Sprachbeschreibung Pascal 2.0

4.4.1 Grundsätzliches

Die Sprache Pascal besitzt eine noch recht kurze Lebensgeschichte. Seit ihrer Vorstellung durch Professor N. Wirth sind noch keine 15 Jahre vergangen. Wegen ihres klaren Stils und nicht zuletzt aufgrund des kompakten Sprachkerns existieren inzwischen zahlreiche Implementierungen auf Großrechenanlagen, Mini- und Mikrocomputern.

Der report

Bis heute existiert jedoch noch kein industrieller Sprachstandard im engsten Sinne des Wortes, wie er z.B. für Fortran und COBOL nicht zuletzt aus kommerziellen Erwägungen geschaffen wurde. Statt dessen beziehen sich alle Implementierungen auf das Buch von Jensen und Wirth »PASCAL, User Manual and Report« [1], in dem die Sprache Pascal definiert wird und eine Implementation der Sprache auf einer CDC 6000 vorgestellt wird.
Jedoch existieren bereits innerhalb dieses Dokumentes einige Inkonsistenzen (siehe auch [3]), so daß eine absolute Kompatibilität zwischen Compilern, die sich an diesem *report* orientieren, nicht gewährleistet ist. Im Gegensatz zu den Sprachen C, Forth, BASIC etc. beschränken sich die Unterschiede jedoch auf eher marginale Details, so daß es also keine essentiell verschiedenen »Pascal-Dialekte« gibt.

Unterschiedliche Implementationen

Nennenswerte Unterschiede gibt es nur in der Realisierung der sogenannten Standardprozeduren und Standardfunktionen. Dabei sind insbesondere die Ein- und Ausgabeoperationen (auf Files) betroffen. Außerdem bieten die meisten Compiler zusätzliche Standardprozeduren. Diese »Verbesserungen« der Sprache machen jedoch Programme, die diese Erweiterungen nutzen, automatisch inkompatibel zu anderen Compilern.

Pascal 2.0

Die vorliegende Realisierung der Sprache *Pascal 2.0* auf dem C 128, einem Heimcomputer, verfolgt folgende Philosophie: Zunächst sollte eine möglichst vollständige und getreue Implementierung des im *report* beschriebenen Sprachumfanges erreicht werden. Dabei mußte jedoch auch die Performanz

auf dem C128 berücksichtigt werden, damit die Entwicklung von Pascal-Programmen nicht zu einer langwierigen, disketten- und nervenaufreibenden Tortur wird. Daher erfolgt der Zyklus Quelltextänderung, Übersetzung und Testlauf menügesteuert und ohne Zugriffe auf externe Speichermedien. Des weiteren sollten auch große Programme mit größeren Datenstrukturen möglich sein, so daß durch *banking* jeweils 64 Kbyte für Daten und übersetzte Programme zur Verfügung stehen. Letztendlich sollten Compiler und Laufzeitsystem ein einfaches *debugging* unterstützen, um den Übergang von einem Interpreter zu einem Compiler nicht allzu schwer zu gestalten.
Andererseits bietet der Computer im C128-Modus unter BASIC eine hervorragende Unterstützung der Hardware (Grafik, Sound, Peripheriegeräte). Um dem Einsteiger in die Sprache Pascal nicht diese attraktiven Möglichkeiten des Computers vorzuenthalten, wurden dem Sprachumfang zahlreiche vordeklarierte Standardroutinen hinzugefügt.
Schließlich existiert noch eine große Palette an *low-level*-Sprachelementen, die die Fähigkeiten der Sprache nach »unten« zur Maschinensprache hin abrunden. Wie in Turbo-Pascal und UCSD-Pascal sind außerdem String-Typen und -Operationen vorhanden.

4.4.2 Implementierungsabhängige Details

Im zweiten Kapitel wurde die Sprache Pascal mit Beispielen schrittweise vorgestellt. In der Dokumentation werden daher nur Details angesprochen, die dort nur ungenau angesprochen werden konnten oder die von anderen Implementierungen der Sprache abweichen.

Namen

Namen im Programmtext

Bezeichner dürfen beliebig lang sein. Dabei sind jedoch nur die ersten 14 Zeichen signifikant. Bei der Eingabe dürfen keine Buchstaben verwendet werden, die mit der Shift-Taste erreicht werden. Dies sind in Abhängigkeit vom Bildschirmmodus Grafikzeichen oder Großbuchstaben.

Datentypen

Wertebereich und Speicherplatz der Datentypen

Die Speicherung der Werte der einzelnen Datentypen erfolgt nach dem folgenden Schema:
Die Standardtypen BOOLEAN und CHAR sowie alle Aufzählungstypen mit weniger als 257 Elementen werden in einem Byte gespeichert. Gleiches gilt für Ausschnitt-Typen der Form A..B, wobei gilt
`ORD(A)>=0 und ORD(B)<=255`
Die übrigen Aufzählungs- und Ausschnitt-Typen, der Typ INTEGER und

Zeiger belegen 2 Byte. Daher gilt für den Wertebereich einer Variablen X des Typs INTEGER:

```
-MAXINT-1 <= X <= MAXINT
```

Dabei ist MAXINT eine vordeklarierte Konstante mit dem Wert 32767. Mengen werden als Bit-Strings dargestellt. Dabei muß für Elemente einer Menge gelten:

```
0 <= ORD(X) <= 95
```

Dementsprechend benötigt jede Mengenvariable 12 Byte. Ein Wert des Typs

```
STRING[Maxlen]
```

benötigt Maxlen+1 Byte. Dabei muß Maxlen im Bereich von 1 bis 255 liegen. Fehlt die Angabe einer Maximallänge, so wird Maxlen=80 angenommen. Diese Strings belegen also 81 Byte.

Speicherverteilung in zusammengesetzten Typen

Nachdem nun der Speicherplatz der elementaren Datentypen bekannt ist, läßt sich der Speicherplatz für zusammengesetzte Objekte nach folgenden Regeln bestimmen:

```
TYPE A: ARRAY [Indextyp] OF Komponententyp;
```

Bei Arrays berechnet sich die Größe des Typs A aus der Größe des Elementtyps multipliziert mit der Kardinalität des Indextyps.

```
TYPE F: FILE OF Komponententyp;
```

Eine File-Variable benötigt neben dem Speicherplatz für den Komponententyp noch 6 Byte für den sogenannten File-Descriptor. Bei Recordtypen berechnet sich die Größe des Verbundtyps aus der Größe der Komponenten. Komponenten werden in der Reihenfolge ihrer Nennung in der Feldliste gespeichert. Bei *varianten* Records werden die verschiedenen Varianten auf demselben Speicherbereich abgebildet. Ein Beispiel soll diese Speicherverteilung illustrieren:

```
R: RECORD
     A: INTEGER;                Offset 0
     B: REAL;                   Offset 2
     CASE C: BOOLEAN OF         Offset 7
          TRUE:
          (S: SET OF CHAR;      Offset 8
           X: 0..9);            Offset 20
          FALSE
          (R: REAL);            Offset 8
   END;
```

Die Variable R benötigt den Speicherplatz für die längste Variante, das sind 21 Byte.

Das Schlüsselwort PACKED wird in *Pascal 2.0* ignoriert.

Typkompatibilität

Zuweisungen sind in Pascal nur zwischen kompatiblen Typen zulässig. Die Typen der aktuellen Variablenparameter bei Prozeduren und Funktionen müssen mit den Typen der formalen Parameter übereinstimmen. Bei verschiedenen Implementierungen existieren jedoch geringfügig unterschiedliche Auffassungen darüber, welche strukturierten Typen gleich sind. In *Pascal 2.0* gelten dabei folgende Regeln:

Kompatibilität zusammengesetzter Typen

Zwei *zusammengesetzte* Variablen besitzen denselben Typ,
- falls sie in einer Variablendeklaration vereinbart oder
- mit dem gleichen Typbezeichner deklariert wurden.

Beispiel:

```
TYPE FELD = ARRAY [0..1] OF INTEGER;
VAR A     : ARRAY [0..1] OF INTEGER;
    B     : ARRAY [0..1] OF INTEGER;
    X,Y   : ARRAY [0..1] OF INTEGER;
    L     : FELD;
    M     : FELD;
```

A und B besitzen nicht den gleichen Typ, obwohl sie eine identische Struktur besitzen. X und Y besitzen den gleichen Typ, da sie in einer Variablendeklaration definiert wurden. L und M besitzen den gleichen Typ, da sie beide mit dem Typbezeichner FELD deklariert wurden. Somit sind folgende Zuweisungen gültig:

```
X:= Y; L:=M
```

aber nicht

```
A:=B; A:=L; M:=X; A:=X
```

Typkompatibilität in String-Ausdrücken

Bei den vordefinierten Typen ist nur der Typ STRING erwähnenswert, da die übrigen Kompatibilitätsregeln bereits in Kapitel 2 genau beschrieben wurden. Ein *String-Ausdruck* ist eine String-Variable, eine String-Konstante, ein Einzelzeichen (vom Typ CHAR), ein ARRAY OF CHAR oder das Ergebnis einer String-Funktion (vordefiniert oder benutzerdefiniert).

Für *String-Zuweisungen* gelten folgende Regeln:

1. Einer String-Variablen kann eine String-Variable beliebiger Maximallänge oder das Ergebnis eines String-Ausdruckes zugewiesen werden. In diesem Fall erfolgt zur Laufzeit ein Test auf Überschreitung der Maximallänge des Ziel-Strings.
2. Einer String-Variablen kann ein String-Ausdruck kleinerer oder gleicher Länge zugewiesen werden.

Für *Wertparameter* des Typs String gelten die gleichen Regeln wie für Zuweisungen an String-Variablen.

Aktuelle *Variablenparameter* des Typs String müssen denselben Typ wie die formalen Parameter besitzen, dabei gelten die oben definierten Regeln für

Pascal 2.0 über die Typkompatibilität zusammengesetzter Typen. Durch den aktiven Kommentar (*$P- *) (siehe Abschnitt 4.4.6.2) wird diese Regel dahingehend gelockert, daß die Maximallänge des aktuellen Parameters größer oder gleich der Länge des formalen Parameters sein muß.

Feldbezeichner

Felder in Records

Die Selektoren müssen im jeweils innersten Bereich eines Records identifizierbar sein. Natürlich dürfen auch gleichnamige Variablen im Programm existieren.

Beispiel:

```
TYPE DEMORECORD =
   RECORD
      A,B: RECORD
             A,B: BOOLEAN
             END;
      C,D: INTEGER;
   END;

VAR  R1  : DEMORECORD;
     A,B : INTEGER;
```

Die obigen Deklarationen sind gültig, da die Sichtbarkeit eines Selektors nur auf den innersten Bereich eines Records beschränkt ist, so daß es keine Konflikte zwischen den einzelnen Namen A und B geben kann. In einer WITH-Anweisung überdecken die Selektornamen die entsprechenden Variablennamen:

Beispiel:

Nach der obigen Deklaration sind folgende Zuweisungen gültig:

```
A:= 3; B:= 4;
WITH R1 DO
  BEGIN
    A.A:= TRUE; A.B:= FALSE;
    WITH B DO
      BEGIN A:= TRUE; B:= FALSE; END;
   END;
```

Records mit Varianten

Keine Prüfung der Gültigkeit einer Variante

Die Varianten werden auf demselben Speicherplatz abgebildet (Details finden sich im vorangegangenen Abschnitt über Datentypen). Die Werte der Variantenwahl (*tagfield*) schränken zur Laufzeit den Zugriff auf die Varianten nicht ein. Es erfolgt also keine Prüfung des *tagfields*.

Beispiel:

```
    TYPE TART = (KARTESISCH, POLAR);
     KOORDINATE =
       RECORD
         CASE ART:   TART OF
           KARTESISCH: (X,Y: REAL);
           POLAR     : (R,PHI: REAL);
       END;
VAR A: KOORDINATE;
```

Die Variante X belegt den Speicherplatz der Variante R, außerdem erfolgt bei der Zuweisung

```
A.ART:= KARTESISCH; A.R:= 20.3;
```

keine Fehlermeldung, obwohl durch die Deklaration des Typs KOORDINATE ausgedrückt wird, daß die Variante R nur für ART = POLAR Gültigkeit besitzt.

WITH-Anweisung

Durch die Verwendung der WITH-Anweisung wird der Sichtbarkeitsbereich des genannten Records geöffnet, so daß nachfolgend Selektoren ohne Nennung der Variablen der WITH-Anweisung verwendet werden können. Wie unter dem Stichwort »Feldbezeichner« beschrieben, überdecken dabei Selektorbezeichner eventuell vorhandene gleichnamige Bezeichner (Variablen, Funktionen,...).

Codeoptimierung mit der WITH-Anweisung

Neben dieser syntaktischen Bedeutung optimiert in *Pascal 2.0* die Verwendung der WITH-Anweisung den erzeugten Code, da sämtliche Adreßberechnungen für die Variable innerhalb der WITH-Anweisung nur genau einmal ausgeführt werden müssen.

Beispiel:

Nach der Deklaration

```
TYPE R: RECORD
          FELD3,FELD4,FELD5: INTEGER;
        END;
VAR A: ARRAY [0..4,0..7] OF ↑R;
```

ist folgende Anweisung kompakter und schneller

```
WITH A[I,J] DO
  FELD5:= FELD5 - FELD3;
```

als die Anweisung

```
A[I,J]↑.FELD5:= A[I,J]↑.FELD5 - A[I,J]↑.FELD3;
```

Sprunganweisung

Keine Sprünge in zusammengesetzte Anweisungen

Die Sprunganweisung (GOTO Label) wurde wie im *report* beschrieben realisiert. Dies bedeutet, daß Sprünge beliebig innerhalb von Blöcken stattfinden können. Hingegen dürfen blockübergreifende Sprünge nur aus einer Prozedur in eine statisch umfassende Prozedur erfolgen. Dabei darf (wie auch im *report* definiert) nicht von außen in ein FOR..DO-, CASE..OF- oder WITH..DO-Statement gesprungen werden. Diese Anweisungen benötigen nämlich zu ihrer korrekten Ausführung Zwischenergebnisse auf dem *stack*.

Beispiel:

```
PROGRAM LABELS (INPUT,OUTPUT);
       LABEL 1,2;
  VAR J: INTEGER;

  PROCEDURE P;
     LABEL 3,4;
    VAR I: INTEGER;
  BEGIN
    GOTO 3;                          (* 1 *)
    FOR I:= 1 TO 3 DO
      BEGIN
        3: WRITELN(I);
        GOTO 4;                      (* 2 *)
      END;
     4: GOTO 1;                      (* 3 *)
   END; (* P *)

BEGIN
  GOTO 3;                            (* 4 *)
  FOR J:= 5 DOWNTO 0 DO
      BEGIN
        2: WRITELN(J);
        IF J=0 THEN GOTO 1;           (* 5 *)
        WRITELN(1/J);
        1: WRITELN; P
      END;
    GOTO 2;                          (* 6 *)
 END.
```

Beispiele für Sprünge

Dieses Programm dient nur Demonstrationszwecken und ist teilweise syntaktisch fehlerhaft. Nachfolgend werden die durchnumerierten Sprunganweisungen besprochen.

1. Von außen darf nicht in eine FOR-Schleife gesprungen werden.
2. Andererseits darf eine FOR-Schleife jederzeit mit einer GOTO-Anweisung verlassen werden.
3. Dieser Sprung aus der Prozedur P heraus ist erlaubt. Zwar wird dadurch in eine FOR-Schleife (FOR J:= 5 DOWNTO 0) gesprungen, jedoch wurde die Prozedur P aus dieser Schleife heraus aufgerufen.
4. In eine Prozedur oder Funktion darf nicht von außen gesprungen werden. Diesen Fehler erkennt bereits der Compiler.
5. Sprünge innerhalb einer FOR-Schleife sind ebenfalls zulässig.
6. Dieser Sprung ist wie (1) nicht erlaubt, da von außen in eine FOR-Schleife gesprungen würde.

Vorwärtsvereinbarungen

Da die Sprache Pascal explizit zur Übersetzung mit *one pass compilern* entworfen wurde, muß innerhalb eines Programmes jeder Name vor seinem ersten angewandten Auftreten ein definierendes Auftreten im Quelltext besitzen. Dabei sind nur die beiden folgenden Ausnahmen definiert worden, um selbstreferenzierende Datenstrukturen und indirekte Rekursionen zu ermöglichen:

Zeigertypen können vor der Deklaration verwendet werden

- Typdeklarationen, die einen Zeiger auf einen noch nicht deklarierten Typ definieren. In diesem Fall muß die Deklaration innerhalb des gleichen Typdeklarationsteils erfolgen.

Beispiel:

```
TYPE ZEIGER = ↑KNOTEN; (* KNOTEN hier noch undefiniert *)
  KNOTEN = RECORD
             INFO: INFOTYP;
             NEXT: ZEIGER;
           END;
```

FORWARD-Deklaration

- Die explizite FORWARD-Vereinbarung von Prozeduren und Funktionen. Die Deklaration des Prozedurrumpfes muß in der gleichen statischen Schachtelungstiefe im selben Block geschehen.

Beispiel:

```
PROGRAM FORWARDDEKLARATIONEN(INPUT,OUTPUT);

PROCEDURE Q(I: INTEGER); FORWARD;

PROCEDURE P(C: CHAR);
BEGIN
  Q(ORD(C));
END; (* P *)
```

```
PROCEDURE Q;
BEGIN
  WRITELN(I);
  IF I>1 THEN P(CHR(I-1));
END; (* Q *)

BEGIN
  Q(5);
END.
```

Case-statement

Syntax für ELSE bei CASE

In *Pascal 2.0* ist die Angabe eines ELSE-Zweiges hinter den Fallmarken möglich (siehe Syntax-Diagramm im Anhang A). Dieser Zweig wird angesprungen, falls der Ausdruck nach dem Schlüsselwort CASE einen Wert liefert, der durch keine Fallmarke erfaßt wird. Wird das Schlüsselwort ELSE nicht angegeben, so wird zur Laufzeit beim Auftreten eines Wertes ohne zugehörige Marke die Fehlermeldung

`NO LABEL IN CASE`

erzeugt. Zwischen dem Schlüsselwort ELSE und dem Schlüsselwort END ist auch eine (evtl. leere) Folge von Anweisungen, getrennt durch Semikola zulässig.

Bit-Operatoren

AND, OR, NOT für INTEGER-Zahlen

Die Operatoren AND, OR und NOT können auch auf Werte vom Typ INTEGER angewendet werden. Es erfolgt dann eine bitweise Verknüpfung der Operatoren durch das logische UND, ODER bzw. eine bitweise Negation der Zahl. Die Operandentypen BOOLEAN und INTEGER dürfen aber nicht gemischt werden. Die Operationen liefern ein Ergebnis des gleichen Typs (INTEGER bzw. BOOLEAN).

Beispiele:

Operation	Ergebnis
TRUE AND FALSE	FALSE
3 AND 4	0
TRUE OR FALSE	TRUE
3 OR 4	7
NOT TRUE	FALSE
NOT 5	–6 (Einerkomplement)
TRUE OR 3	– Operation nicht zulässig –

Absolut adressierte Variablen

Absolute Variablen für die Systemprogrammierung

Bei einer Variablendeklaration in *Pascal 2.0* ist die explizite Angabe einer absoluten Adresse für die Variable möglich. Für diese Variablen wird dann kein Speicher auf dem *stack* reserviert, sondern die Variable bleibt (auch bei rekursiven Aufrufen) an die bei der Deklaration vergebene Adresse gebunden.
Ein Anwendungsgebiet ist die gemeinsame Verwendung von Variablen aus Maschinenprogrammen und Betriebssystemadressen in Pascal-Programmen. Der Speicherplatzbedarf in Bytes für jeden Variablentyp läßt sich mit der Funktion SIZEOF berechnen.
Auf dem C 128 sind absolute Variablen nur in Bank 1 möglich. Ausgenommen hiervon sind Variablen in der sogenannten *common area*, die von der Adresse $0000 bis $0400 reicht. Um Speicheradressen in anderen Bereichen anzusprechen, müssen die Prozeduren POKE und PEEK verwendet werden.
Beispiel:
Möchten Sie z.B. die Eckpunkte des laufenden Windows feststellen, so können Sie auf die in der Zero-Page gespeicherten Werte zurückgreifen:

```
VAR UNTEN : BYTE [228];
    OBEN  : BYTE [229];
    LINKS : BYTE [230];
    RECHTS: BYTE [231];
```

Mit diesen Variablen können Sie dann wie gewohnt arbeiten:

```
WRITELN('FENSTER VON', LINKS, ' BIS', RECHTS);
LINKS:= LINKS+1; RECHTS:= LINKS+4;
```

In diesem speziellen Fall sollten Sie jedoch nach einer Veränderung der Betriebssystemvariablen für das aktive Fenster zunächst das Fenster neu initialisieren:

```
WRITE(CHR(19));
```

String-Operationen

Operationen mit Strings in Pascal 2.0

Die Deklaration von String-Variablen wurde bereits in dem Abschnitt über Datentypen besprochen. Neben den in Abschnitt 4.4.4 vorgestellten String-Prozeduren und -Funktionen sind noch folgende Operationen mit Strings zulässig, die sich an die Operationen mit skalaren Objekten anlehnen:
Beliebige String-Ausdrücke (String-Variablen, -Konstanten, Funktionsergebnisse, ARRAYs OF CHAR und Einzelzeichen vom Typ CHAR) können mit den relationalen Operatoren =, <>, >=, <=, > und < verglichen werden. Das Ergebnis des Vergleiches ist vom Typ BOOLEAN und hängt vom zugrundeliegenden, Commodore-eigenen ASCII-Zeichensatz ab (siehe Anhang A im BASIC-Handbuch).

Beispiel:

```
'OTTO' > 'ANNA'
'X '   <> 'X'
'X'    <> 'x'
'ANNA' < 'ANNABELLE'
```

Schließlich können beliebige String-Ausdrücke durch den Additionsoperator + verkettet (konkateniert) werden:

Beispiel:

```
VAR S: STRING;
    N: INTEGER;

S:= 'Gemahlin'; N:= 5;
S:= 'Hochzeitsgrüße an Sie und Ihre' + STR(N:0:0) +
    '. ' + S;
```

Funktionsergebnisse

Lange Funktionsergebnisse

In *Pascal 2.0* kann das Ergebnis einer Funktion einen beliebigen, also auch zusammengesetzten Typ besitzen. Dieser Typ muß durch einen Typbezeichner definiert werden:

Beispiel:

```
PROGRAM LONGFUNCTIONRESULTS(INPUT,OUTPUT);
TYPE RANDOMRECORD: RECORD
                     X,Y: INTEGER;
                   END;
VAR R: RANDOMRECORD;
    S: STRING;

FUNCTION RANDOM2: RANDOMRECORD;
BEGIN
 RANDOM2.X:= INT(RANDOM(0)*32767);
 RANDOM2.Y:= INT(RANDOM(0)*32767);
END; (* RANDOM2 *)

FUNCTION RANDOMSTRING(SIZE: INTEGER): STRING;
 VAR S: STRING;
BEGIN
 S:='';
 REPEAT
  S:= S+CHR(INT(RANDOM(0)*20)+ORD('A'));
 UNTIL LENGTH(S)=SIZE;
 RANDOMSTRING:= S;
END; (* RANDOMSTRING *);
```

```
BEGIN
 R:= RANDOM2;
 S:= RANDOMSTRING(40);
END.
```

Typbindungsoperatoren

Wie in der Sprache C sind in *Pascal 2.0* explizite Typumwandlungen möglich. Durch die Nennung eines Typnamens wird ein Wert auf dem *stack* an diesen Typ angepaßt:

```
PROGRAM TYPEBINDING(INPUT,OUTPUT);
TYPE MONTH = (JAN, FEB, MRZ, APR, MAI, JUN, JUL, AUG, SEP,
              OKT, NOV, DEZ);
     VAR M: MONTH;
         I: INTEGER;
         P: ↑INTEGER;
BEGIN
  M:= MONTH(3);
  M:= MONTH(I);
  M:= MONTH(ORD(SUCC(MONTH)) MOD (ORD(DEZ)+1));
  I:= INTEGER(M);
  I:= INTEGER(P);
END.
```

Typbindung ohne Prüfung durch den Compiler

Bei dieser Typanpassung werden **keinerlei** Prüfungen durchgeführt. Jedoch sollten Typanpassungen nur zwischen skalaren Werten und Zeigern durchgeführt werden.

Erweiterungen der Syntax

Freie Reihenfolge der Deklarationsteile

Im *report* ist die Reihenfolge der Deklarationsteile folgendermaßen festgelegt: Konstanten-, Typ-, Variablen- und dann Prozedurdeklarationen. In *Pascal 2.0* können diese Teile in einer beliebigen Reihenfolge und auch mehrmals aufgeführt werden. Damit kann man z.B. Variablendeklarationen, die nur im Rumpf des Hauptprogrammes sichtbar sein sollen, *hinter* die Prozedurdeklarationen stellen. Insbesondere bei der Benutzung von Include-Files als abgeschlossene *Funktionsmodule* kann man *private* Variablen und Typen dieser Module zusammen mit den *exportierten* Prozeduren aufführen, ohne in Konflikte mit der Syntax zu geraten.

Sonderzeichen in Stringkonstanten

Schließlich wurde die Syntax für String-Konstanten erweitert. Damit ist es möglich, String-Konstanten zu definieren, die länger als eine Quelltextzeile sind und die Steuerzeichen enthalten. Außerdem können String-Konstanten wahlweise durch Anführungszeichen »″« oder Hochkommata »'« eingeschlossen werden. Dabei darf ein String das entsprechende Zeichen *nicht* enthalten.

Beispiel:

```
PROGRAM SYNTAX(INPUT,OUTPUT);
CONST RVSON  = 18;
      RVSOFF = 146;
VAR   X: INTEGER;
CONST SCONST1 = 'Dieser String ist länger als eine'
                "einzelne Bildschirmzeile";
      SCONST2 = #13 #7 #7 'Dieser String enthält '
                # RVSON 'Steuerzeichen' #RVSOFF ;
      SCONST3 = "In Anführungszeichen ist ein  Hoch"
           "komma (') erlaubt ";
BEGIN
  WRITE(#147, SCONST1, SCONST2);
END.
```

4.4.3 Reservierte Wortsymbole

Keine neuen Wortsymbole

In *Pascal 2.0* existieren nur die im *report* definierten Wortsymbole. Daher kann es nicht zu Konflikten zwischen neu definierten Wortsymbolen und benutzerdefinierten Namen kommen. Die folgende Liste enthält alle Wortsymbole der Sprache Pascal, die nicht als Namen (z.B. für Variablen) verwendet werden dürfen:

and	file	not	to
array	for	of	type
begin	forward	or	until
case	function	packed	var
const	goto	procedure	while
div	if	program	with
do	in	record	
downto	label	repeat	
else	mod	set	
end	nil	then	

4.4.4 Vordefinierte Namen

Änderung der Bedeutung eines vordefinierten Namens

Im Gegensatz zu den oben aufgelisteten Wortsymbolen besitzen die in diesem Abschnitt vorgestellten Namen eine vordefinierte Bedeutung, die jedoch durch Deklarationen verändert werden kann.

Beispiel:

```
TYPE STRING = ARRAY[0..ALEN] OF CHAR;
```

In diesem Beispiel wird dem in *Pascal 2.0* vordefinierten Typbezeichner STRING eine neue Bedeutung gegeben. Im Sichtbarkeitsbereich der obigen

Deklaration wird deshalb mit dem Namen STRING ein ARRAY OF CHAR und nicht ein String variabler Länge bezeichnet.

4.4.4.1 Namen für Konstanten

FALSE, TRUE — Konstanten des Typs BOOLEAN. Es gilt FALSE< TRUE, da ORD(FALSE)=0 und ORD(TRUE)=1 ist.

MAXINT — Konstante des Typs INTEGER mit dem Wert 32767. MAXINT ist die größte darstellbare ganze Zahl.

4.4.4.2 Namen für Typen

BOOLEAN — Vordefinierter Aufzählungstyp mit den Werten FALSE und TRUE.

INTEGER — Ganze Zahlen im Bereich von –MAXINT–1 bis MAXINT.

CHAR — Einzelne Zeichen.

REAL — Reelle Zahlen. Der Wertebereich ist wie in BASIC definiert.

TEXT — Dieser File-Typ ist definiert durch

```
TYPE TEXT = FILE OF CHAR
```

STRING — Dieser Typbezeichner spielt eine Sonderrolle, da er eine ganze »Klasse« von Typen umfaßt: Durch die Angabe einer Feldlänge in eckigen Klammern nach dem Typnamen STRING kann die Maximallänge einer String-Variablen festgelegt werden.

BYTE — Dieser Typbezeichner entspricht dem Unterbereichstyp:

```
TYPE BYTE = 0..255
```

Werte des Typs BYTE belegen nur 1 Byte Speicherplatz.

4.4.4.3 Namen für Variablen

INPUT — Diese Variable vom Typ TEXT (siehe oben) ist beim Start eines Programmes automatisch mit dem Standardeingabemedium (der Tastatur) verbunden.

OUTPUT — Diese Variable vom Typ Text repräsentiert das Standardausgabemedium (den Bildschirm).

BANK — Die Operationen POKE, PEEK und SYS benötigen neben einer Adresse noch die Angabe der anzusprechenden Bank. In *Pascal 2.0* wird die für diese Operationen gültige Bank mit der variablen Bank verwaltet. Einerseits kann mit Zuweisungen die aktive Bank umgeschaltet werden, andererseits kann in Ausdrücken die Variable zum Bestimmen der momentan aktiven Bank bestimmt werden. Beim Start eines Pascal-Programmes wird die variable Bank mit dem

Wert 15 vorbelegt. Sinnvolle Werte für Bank liegen im Bereich zwischen 0 und 15:

BANK	Konfiguration:
0	nur RAM 0
1	nur RAM 1
2	nur RAM 2 (bzw. RAM 0)
3	nur RAM 3 (bzw. RAM 1)
4	Internes ROM, RAM 0, I/O Area
5	Internes ROM, RAM 1, I/O Area
6	Internes ROM, RAM 2, I/O Area
7	Internes ROM, RAM 3, I/O Area
8	Externes ROM, RAM 0, I/O Area
9	Externes ROM, RAM 1, I/O Area
10	Externes ROM, RAM 2, I/O Area
11	Externes ROM, RAM 3, I/O Area
12	Externes ROM (L), I/O, Rest RAM 0
13	Externes ROM (L), I/O, Rest RAM 1
14	BASIC, Kernal, CHARROM, Rest RAM 0
15	BASIC, Kernal, I/O, Rest RAM 0

BANK-Switching und die Common-area

Durch die Wahl des RCR (*ram configuration register*) der MMU ist im Speicherbereich $0000 bis $0400 eine *common-area* deklariert. In diesem Bereich kann daher nur RAM in Bank 0 adressiert werden.

Beispiel:

```
BANK:=0; SYS(-195); (* Kaltstart ausfuehren *)
WRITELN('MOMENTAN IST BANK', BANK, ' AKTIVIERT');
WRITELN(PEEK(32767));
```

4.4.4.4 Prozeduren für dynamische Objekte

Verwaltung des heap

Die dynamischen Objekte werden nicht wie »normale« Variablen automatisch auf dem *stack* verwaltet, sondern besitzen einen eigenen Speicherbereich, den *heap* (siehe auch Abschnitt 4.3.4). Reicht zur Laufzeit des Programmes der Speicher auf dem *heap* nicht mehr aus, so wird das Programm mit folgender Fehlermeldung beendet:

```
HEAP OVERFLOW ERROR
```

Der *heap* ist als *last-in-first-out*-Speicher organisiert: Mit NEW wird am »unteren« Ende des *heap* Speicherplatz reserviert, während mit RELEASE Speicherplatz auch am »unteren« Ende freigegeben wird. Dieses Ende des *heap* wird durch einen sogenannten *heappointer* markiert.

Die Befehle MARK und RELEASE ersetzen in *Pascal 2.0* (wie z.B. auch in Turbo-Pascal) die in anderen Dialekten vorhandene Prozedur DISPOSE.

NEW(Zeigervariable)

Ein Aufruf dieser Standardprozedur stellt Speicherplatz für eine (neue) dynamische Variable zur Verfügung. Die Zeigervariable wird mit der Adresse des neuen Objektes initialisiert.

MARK(Zeigervariable)
RELEASE(Zeigervariable)

Rückgewinnung unbenutzten Speicherplatzes

Um den mit einer Folge von Aufrufen der Prozedur NEW reservierten Speicherplatz wieder freizugeben, verwendet man die Prozeduren MARK und RELEASE. *Vor* den NEW-Operationen wird zunächst der Wert des *heappointer* mit der Prozedur MARK in einer Zeigervariablen gespeichert. Um nun den reservierten Speicherplatz wieder freizugeben, genügt der Aufruf der Prozedur RELEASE mit dieser Zeigervariablen. Dadurch wird der *heappointer* auf den mit MARK markierten Wert zurückgesetzt. Durch diese Art der Speicherverwaltung ist es *nicht* möglich, daß Lücken im *heap* existieren.
Um nicht zusammenhängende Speicherbereiche freizugeben, müssen explizit Freispeicherlisten (siehe Kapitel 2) verwaltet werden.

Beispiel:

```
PROGRAM DYNAMIC(INPUT,OUTPUT);

VAR A,B,C,X,Y: ↑INTEGER;

BEGIN
  MARK(Y);              (* 1 *)
  NEW(A); NEW(B);
  MARK(X);              (* 2 *)
  NEW(C);               (* 3 *)
  RELEASE(X);           (* 4 *)
  RELEASE(Y);           (* 5 *)
END.
```

Die folgende Abbildung zeigt die durch diese Prozeduraufrufe entstehende Speicherverteilung auf dem *heap*:

1. Der leere *heap* wird mit Y markiert.
2. Es wird Speicherplatz für A↑ und B↑ reserviert, und die Größe des *heap* wird mit X markiert.
3. Es wird zusätzlich Speicherplatz für C reserviert.
4. Der Speicher wird auf seinen mit X markierten Zustand zurückgesetzt. Dadurch wird der Speicherplatz der Variablen C↑ freigegeben.

5. Alle seit der Operation MARK(Y) mit NEW angelegten dynamischen Variablen werden wieder gelöscht. Damit wird insbesondere der Speicherplatz für A↑ und B↑ wieder freigegeben.

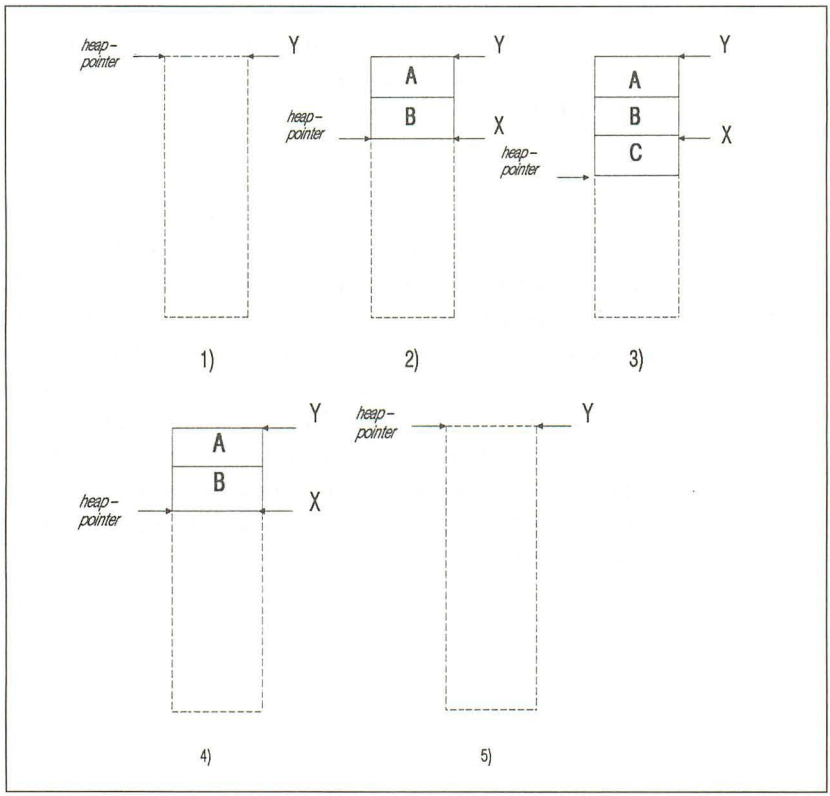


Bild 118: Wirkung der Prozeduren NEW, MARK und RELEASE

4.4.4.5 Ein- und Ausgabeoperationen

File-Operationen in Pascal 2.0

File-Variablen in *Pascal 2.0* werden durch Dateien unter dem Betriebssystem des C 128 realisiert. Alle Befehle werden im Zusammenhang in Abschnitt 4.4.5 besprochen. Komplette Programmbeispiele finden sich in Kapitel 2.16, 2.17 und 3.1.

OPEN(file-Variable, Geräteadresse, Sekundäradresse, [Name])

Standard-operationen mit File-Variablen

Das genannte File wird zum Lesen, Schreiben oder Modifizieren eröffnet. Intern wird dem angegebenen File eine freie logische File-Nummer zugewiesen, unter der die File-Variable vom Betriebssystem identifiziert wird. Mit dieser Nummer wird ein Aufruf der OPEN-Routine des Betriebssystems durchgeführt. Anschließend ist EOF(File-Variable)=FALSE und der Wert File-Variable↑ ist undefiniert.

File-Variable	Variable des Typs FILE OF ...
Geräteadresse	Integerausdruck, der eine positive Geräteadresse im Bereich 0 bis 255 liefert.
Sekundäradresse	Integerausdruck, der eine Sekundäradresse für das Gerät liefert. Möchten sie keine Sekundäradresse spezifizieren, so wählen Sie als Sekundäradresse 0.
Name	Dieser Ausdruck vom Typ String liefert den Namen, unter dem dieses File gespeichert wird. Der Name kann auch entfallen.

CLOSE (file-Variable)

Ein zuvor mit OPEN zum Schreiben oder Lesen eröffnetes File *muß* am Ende der Verarbeitung mit CLOSE geschlossen werden. Dies ist insbesondere vor einem erneuten OPEN erforderlich. Die mit OPEN reservierte logische File-Nummer wird wieder freigegeben. Das Betriebssystem kann maximal 10 gleichzeitig eröffnete Files verwalten.
Nach CLOSE sind keine weiteren PUT-, GET-, READ- oder WRITE-Operationen mit der File-Variablen zulässig. Normalerweise erscheint in diesem Fall die Fehlermeldung

```
FILE NOT OPEN ERROR
```

Wurde jedoch inzwischen die mit CLOSE freigewordene logische Filenummer vom Betriebssystem erneut vergeben, so erfolgen die Ausgaben auf dieses neu eröffnete File.

EOF(file-Variable)

Das Ergebnis dieser Funktion (*end of file*) ist vom Typ BOOLEAN. EOF(F) ist TRUE, falls beim Lesen der Datei F das File-Ende erreicht wurde. Beim Schreiben ist EOF(F) immer TRUE. EOF alleine ist die Abkürzung für EOF(INPUT). In diesem Abschnitt wird für jede File-Operation beschrieben, wie der Funktionswert EOF verändert wird.

STATUS(file-Variable)

Systemnahe Informationen über ein File

Diese Funktion existiert nur in *Pascal 2.0* und besitzt den Ergebnistyp INTEGER. *Pascal 2.0* orientiert sich bei File-Operationen stark an den Vorgaben des *report* und verbirgt vor dem Benutzer die Realisierung der einzelnen File-Operationen. Jedoch kann es in Spezialfällen sinnvoll sein, direkt auf die Betriebssystemebene herabzusteigen. Daher ermöglicht die Funktion STATUS, direkt die Ergebnisse der Ein- und Ausgabeoperationen zu kontrollieren: Das niederwertige Byte (L-Byte) des Funktionsergebnisses STATUS(F) liefert den Status bei der letzten E-/A-Operation (READ, WRITE, GET, PUT) auf dem File F. Dieser Wert ist wie für die BASIC-Variable ST definiert. Einzel-

heiten finden sich in den Handbüchern zum C128. Wichtig ist, daß der Wert von STATUS(F) *unabhängig* von E-/A-Operationen auf anderen Files ist.
Das H-Byte von STATUS(F) enthält die logische File-Nummer, unter der das Betriebssystem das File verwaltet. Dieser Wert ist nur zwischen OPEN(F,...) und CLOSE(F) definiert.
STATUS alleine ist die Kurzform für STATUS(INPUT).

Beispiel:

```
PROGRAM ST(INPUT,OUTPUT);
  VAR F,G: TEXT;
      LINE: STRING;
BEGIN
  OPEN(F,8,3,'DATAREAD,S,R');
  OPEN(G,8,5,'DATAWRITE,S,W');
  WRITELN('F hat die logische File-Nummer',
          HBYTE(STATUS(F)));
  WRITELN('G hat die logische File-Nummer',
          HBYTE(STATUS(G)));
  REPEAT
     READ(F,LINE);
     WRITE(G,LINE);
  UNTIL EOF(F);
  IF (STATUS(F) AND 2)<>0 THEN
     WRITELN('Zeitablauf beim Lesen');
  IF (STATUS(F) AND 128)<>0 THEN
     WRITELN('Gerät nicht angeschlossen');
  IF (STATUS(F) AND 64)<>0 THEN
     WRITELN('Datenende erreicht');
  CLOSE(F); CLOSE(G);
END.
```

EOLN(file-Variable)

Diese Standardfunktion besitzt den Ergebnistyp BOOLEAN. Der Parameter File-Variable muß ein FILE OF CHAR, also ein Textfile sein. EOLN(F) liefert den Wert TRUE, falls beim Lesen mit READ(F, ...) das Zeilenende erreicht wurde.
Das Zeilenende in Pascal-Textdateien wird mit dem ASCII-Zeichen CR (*carriage return*, CHR(13)) markiert. Ist jedoch EOLN(F)=TRUE, so enthält die Puffervariable F↑ ein Leerzeichen (' ' = CHR(32)).
Wie bei EOF und STATUS entspricht EOLN alleine dem Prozeduraufruf EOLN(INPUT), so daß man den Zustand des Standardeingabe-Files INPUT erhält.

PUT(file-Variable)

Operationen mit beliebigen Files

Nachdem ein File zum Schreiben eröffnet wurde, kann man es sequentiell mit Werten füllen. Der Aufruf PUT(F) überträgt den Inhalt der Puffervariablen F↑ Byte für Byte auf das File F. Anschließend gilt EOF(F)=TRUE.

GET(file-Variable)

Um mit PUT erstellte Files zu lesen, verwendet man nach dem Eröffnen mit OPEN die Standardprozedur GET. Der Aufruf GET(F) füllt die Puffervariable F↑ Byte für Byte mit dem Inhalt des Files. Außerdem wird der Lesezeiger im File um eine Position weitergesetzt. Ist nach dem Aufruf EOF(F)=TRUE, so ist der Wert F↑ undefiniert.
Während das Betriebssystem des C128 beim letzten übertragenen Byte einer Datei die Variable ST verändert, wird in Pascal EOF(F) erst dann TRUE, wenn ein Wert *hinter* dem File-Ende gelesen werden soll.

READIN(file-Variable)

Operationen mit Text-Files

Mit READLN(F) werden Zeichen bis zum Ende der Eingabezeile im Text-File F überlesen. Ist EOLN(F) beim Aufruf von READLN(F) bereits TRUE, so werden *keine* Zeichen gelesen.

READ(file-Variable, Parameter {,Parameter})

Von dem durch die File-Variable F angegebenen Text-File werden Werte verschiedener Typen gelesen. Die Eingabe hängt vom Typ der Parameter ab.

- Der Parameter ist eine Variable vom Typ CHAR. Dann wird ein einzelnes Zeichen vom File gelesen und der Variablen zugewiesen. Ist das Zeichen ein Zeilenendezeichen, so wird EOLN(F) TRUE. Jedoch wird der Variablen in diesem Fall ein Leerzeichen; (' ' = CHR(32)) zugewiesen.
- Der Parameter ist eine Variable vom Typ INTEGER oder REAL. In diesem Fall werden zunächst Leerzeichen und Zeilentrennzeichen überlesen. Die nachfolgenden Zeichen werden bis zum nächsten Leerzeichen oder Zeilenende gelesen und als Zahl interpretiert (wie bei der Funktion VAL). Ist der Parameter eine Variable vom Typ INTEGER, so wird die Zahl noch mit der Funktion INT angepaßt.
- Ist der Parameter eine String-Variable, so werden Zeichen bis zum nächsten Zeilenende eingelesen und in der String-Variablen gespeichert. Übersteigt die Anzahl der gelesenen Zeichen die Maximallänge des Strings, so wird das Programm mit einer Fehlermeldung beendet.

  Eingabe leerer Strings

  Beim Einlesen vom Standard-Eingabemedium INPUT erlaubt das Betriebssystem keine Eingabe von Strings der Länge 0. Wird die Eingabe nur durch die Return-Taste beendet, so erhält man einen String der Länge 1, der nur aus einem Leerzeichen besteht.

Werden mehrere Parameter genannt, so erfolgt die Eingabe direkt aufeinanderfolgend. EOF(F) ist nach READ(F,...) TRUE, falls das letzte Zeichen des Files gelesen wurde. Die Puffervariable F↑ enthält in jedem Fall das letzte gelesene Zeichen. War dies ein Zeilenendezeichen (CR), so liefert die Funktion EOLN(F) den Wert TRUE. Die Puffervariable F↑ enthält in diesem Fall jedoch ein Leerzeichen.
Die Angabe

```
READLN(File-Variable, Parameter {,Parameter})
```

entspricht der Anweisungsfolge

```
READ(File-Variable, Parameter {,Parameter});
READLN(File-Variable);
```

WRITELN(file-Variable)

Mit dem Prozeduraufruf WRITELN(F) wird auf dem File F ein Zeilenendezeichen (CR, *carriage return*, CHR(13)) geschrieben. Anschließend gilt: EOF(F)=TRUE und EOLN(F)=TRUE.

WRITE(file-Variable, Parameter {,Parameter})

Es können beliebig viele, durch Kommata getrennte Parameter genannt werden. Das Ergebnis der Ausgabe hängt von den Typen der Ausdrücke ab. Bei jedem Parameter X kann hinter einem Doppelpunkt eine *Feldgröße* N angegeben werden. Dies ist ein Ausdruck vom Typ INTEGER, der die Mindestanzahl an Zeichen angibt, die gedruckt werden. Ist der auszugebende Wert jedoch länger als die Feldgröße, so wird die Feldgröße ignoriert und der Wert mit minimaler Länge ausgegeben.

Wirkung der Feldgrößen bei WRITE

`WRITE(X)` oder `WRITE(X:N)`

- Ist der Ausdruck X vom Typ CHAR, so wird ein einzelnes Zeichen ausgegeben. Ist eine Feldgröße N spezifiziert, so wird das Zeichen rechtsbündig in einem Feld von N Stellen ausgegeben.
- Der Ausdruck X ist vom Typ INTEGER. Dann wird die ganze Zahl rechtsbündig in einem Feld von N Stellen ausgegeben. Vor einer positiven Zahl steht ein Leerzeichen und kein »+«.
- Der Ausdruck X ist ein String-Ausdruck. In diesem Fall wird der String in einem Feld von N Stellen rechtsbündig ausgegeben.
- Ein Ausdruck X vom Typ REAL wird in Exponentialdarstellung mit 8 Nachkommastellen gedruckt. Fehlt die Angabe einer Feldgröße N, so werden 15 Zeichen gedruckt. Ist der Wert N größer als 15, so wird die Exponentialdarstellung in einem N-stelligen Feld rechtsbündig gedruckt.

Nachkommastellen bei reellen Zahlen

- Bei einem reellen Ausdruck X ist jedoch auch ein zweiter Formatierungsparameter M zulässig, der die Anzahl der Nachkommastellen festlegt:
  `WRITE(X:N:M)`
  Ist M > =0, so wird die Zahl X rechtsbündig in einem Feld aus N Stellen gedruckt. Dabei wird eine Festkommadarstellung mit M Nachkommastellen gewählt.
  Ist M < 0, so wird eine Exponentialdarstellung mit ABS(M) Nachkommastellen verwendet. N definiert wiederum die Feldgröße, in der die gesamte Zahl rechtsbündig ausgegeben wird.
  Beispiele:

```
WRITE(-3.3);            -3.30000000E+00
WRITE(-3.3:20);         -3.30000000E+00
WRITE(-3.3:20:4);                  -3.3000
WRITE(-3.3:20:0);                       -3
WRITE(-3.3:8:-3);       -3.300E+00
```

GETKEY(Variable)

Die Variable muß vom Typ CHAR sein. Es wird das nächste Zeichen des Tastaturpuffers gelesen. Wurde keine Taste gedrückt, so wird das Zeichen CHR(0) gelesen. In keinem Fall wird auf eine Tastatureingabe gewartet. Außerdem erscheint kein blinkender Cursor am Bildschirm.

Abfrage der Run/Stop-Taste

Da die Run/Stop-Taste durch das Pascal-System nicht abgefragt wird, kann man auch explizit die Betätigung der Run/Stop-Taste (mit dem Code CHR(3)) prüfen:

Beispiel:

```
PROGRAM GETIT(INPUT,OUTPUT);
  VAR C: CHAR;
BEGIN
  REPEAT
    GETKEY(C); WRITE(C);
  UNTIL C = CHR(3);
  WRITELN('STOP-TASTE GEDRUECKT');
END.
```

KEYPRESSED

Diese Funktion liefert ein Ergebnis vom Typ BOOLEAN. Ist eine Taste gedrückt worden, so wird der Wert TRUE geliefert. Die Funktion ließe sich folgendermaßen durch GETKEY ersetzen:

```
FUNCTION KEYPRESSED: BOOLEAN;
  VAR C: CHAR;
BEGIN
  GETKEY(C);
  KEYPRESSED:= C<> CHR(0);
END;
```

4.4.4.6 Arithmetische Funktionen

Funktionen mit skalaren Typen

ORD(Ausdruck)

Diese Funktion liefert als Ergebnis eine Zahl vom Typ INTEGER, die die Position des Parameters im Wertebereich angibt (0,1,2,...). Der Ausdruck muß einen skalaren Typ besitzen (also nicht REAL).

CHR(Ausdruck)

Diese Funktion liefert ein Zeichen mit dem ASCII-Code des INTEGER-Ausdruckes.

SUCC(Ausdruck)

Die Funktion liefert den Nachfolger im Wertebereich. Der Ausdruck muß einen skalaren Typ besitzen. Eine Prüfung auf Bereichsüberschreitung erfolgt nur, wenn bei der Übersetzung die Option (*$R+ *) des Compilers aktiv war.

PRED(Ausdruck)

Die Funktion liefert den Vorgänger im Wertebereich. Sonst gelten die gleichen Nebenbedingungen wie sie bei SUCC beschrieben wurden.

Funktionen für reelle und ganze Zahlen

ODD(Ausdruck)

Diese Funktion liefert den Wert TRUE, falls der Ausdruck vom Typ INTEGER ungerade ist:

Beispiel:

```
ODD(3) = TRUE
ODD(-1) = TRUE
ODD(0) = FALSE
```

ABS(Ausdruck)

Der Absolutbetrag des REAL- oder INTEGER-Argumentes wird berechnet. Das Ergebnis ist vom selben Typ wie der aktuelle Parameter.

INT(Ausdruck)

Der reelle Ausdruck wird in die nächstkleinere INTEGER-Zahl umgewandelt.

TRUNC(Ausdruck)

Bei dem reellen Ausdruck werden Nachkommastellen abgeschnitten, um eine INTEGER-Zahl zu erhalten.

ROUND(Ausdruck)

Der reelle Ausdruck wird zur nächsten ganzen Zahl gerundet. Bei INT, TRUNC und ROUND erfolgt bei einer Überschreitung des Bereiches für INTEGER-Zahlen (–MAXINT–1 bis MAXINT) ein Programmabbruch mit Fehlermeldung.

Beispiele:

```
ROUND( 3.2) = 3   TRUNC( 3.2) = 3   INT( 3.2) = 3
ROUND( 3.7) = 4   TRUNC( 3.2) = 3   INT( 3.2) = 3
ROUND(-3.2) = 3   TRUNC(-3.2) = 3   INT(-3.2) = 4
ROUND(-3.7) = 4   TRUNC(-3.2) = 3   INT(-3.2) = 4
```

SQRT(X), LN(X), EXP(X), SIN(X), COS(X), ARCTAN(X)

Diese Funktionen liefern, wie im *report* beschrieben, zu den reellen Argumenten X die Werte der Funktionen Quadratwurzel, natürlicher Logarithmus, Exponentialfunktion, Sinus, Cosinus und Arcustangens. Das Ergebnis ist natürlich vom Typ REAL.

TAN(X)

Diese Funktion ist nicht im *report* aufgeführt. Sie liefert zu der reellen Zahl X den Tangens in Bogenmaß (ebenfalls vom Typ REAL).

SQR(X)

Diese Funktion quadriert die reelle Zahl X. Das Ergebnis ist vom Typ REAL. Bei Zahlen vom Typ INTEGER sollte man statt dessen X*X schreiben.

POWER(X,Y)

Diese Funktion (nicht im *report* vorhanden) liefert zu den Ausdrücken X und Y vom Typ REAL den Wert X hoch Y vom Typ REAL. Die Berechnung erfolgt intern über die Logarithmus- und Exponentialfunktion. Daher sollte die Funktion POWER in laufzeitkritischen Anwendungen für kleine ganzzahlige Y durch geeignete Multiplikationen ersetzt werden.

RANDOM(X)

Zufallszahlen

Es wird eine Pseudo-Zufallszahl R aus dem Bereich

$0 <= R < 1$

erzeugt. Der Algorithmus zur Berechnung von R hängt vom Vorzeichen des INTEGER-Ausdruckes X ab:

- Ist X > 0, so wird bei aufeinanderfolgenden Aufrufen der Funktion RANDOM eine feste »Zufallsreihe« durchlaufen. Diese ist unabhängig von X und wird bei jedem Einschalten des Rechners wieder neu begonnen.
- Ist X < 0, so wird für jeden Wert X eine andere Zufallsreihe erzeugt. Gleiche Argumente X liefern dieselbe Reihe.
- Ist X = 0, so wird die Zufallszahl aus einer Verknüpfung der Werte verschiedener systeminterner Zählregister erhalten.

4.4.4.7 Vordefinierte String-Prozeduren

Exakte Definition der String-operationen

In Anlehnung an die String-Operationen anderer Pascal-Implementierungen wurden in *Pascal 2.0* folgende vordefinierte Funktionen und Prozeduren aufgenommen:

INSERT(String-Ausdruck,String-Variable,Position)

Der String-Ausdruck wird in die String-Variable ab der genannten Position eingefügt. Liegt die Position nicht im Bereich 1..255, so wird das Programm mit der Fehlermeldung »ILLEGAL QUANTITY ERROR« abgebrochen. Überschreitet die Länge des Ergebnis-Strings die für die String-Variable deklarierte Maximallänge, so wird die Fehlermeldung »STRING OVERFLOW ERROR« ausgegeben. Befindet sich die Position hinter der aktuellen Länge der String-Variablen, so werden die beiden Strings konkateniert:

Beispiel:

```
VAR S,T: STRING;
S:= 'Computer'; INSERT('Home-',S,1);
T:= 'kurz '; INSERT('und bündig',T,40);
```

Nach diesen Anweisungen besitzen die Strings die Werte 'Home-Computer' sowie 'kurz und bündig'.

DELETE(String-Variable,Position,Länge)

Mit dieser Prozedur wird aus der String-Variablen ab der genannten Position ein Teil-String der angegebenen Länge entfernt. Ist die Position außerhalb des Bereiches 1 bis 255 oder die Länge außerhalb des Bereiches 0 bis 255, so wird die Fehlermeldung »ILLEGAL QUANTITY ERROR« ausgegeben. Liegt die Position hinter dem Ende des Strings, so werden keine Zeichen gelöscht. Ist die Summe aus Position und Länge größer als die Länge des Strings, so werden nur Zeichen innerhalb des Strings gelöscht.

CONCAT(String-Ausdruck, String-Ausdruck {,String-Ausdruck})

Die Ergebnisse der String-Ausdrücke werden konkateniert. Entstehen dabei Strings, die länger als 255 Zeichen sind, so wird die Programmausführung mit

der Fehlermeldung »STRING OVERFLOW ERROR« beendet. Diese Funktion existiert nur, um kompatibel mit anderen Pascal-Implementierungen zu bleiben. Einfacher und übersichtlicher erreicht man eine Konkatenation von Strings mit dem »+«-Zeichen.

Beispiel:

```
S:=CONCAT('Pascal', ' 2.0') ist äquivalent zu
S:= 'Pascal' + ' 2.0'
```

COPY(String-Ausdruck,Position,Länge)

Mit dieser Funktion wird aus dem String-Ausdruck ein String ab der genannten Position mit der gegebenen Länge isoliert. Ist die Position außerhalb des Bereiches 1 bis 255 oder die Länge außerhalb des Bereiches 0 bis 255, so wird die Fehlermeldung »ILLEGAL QUANTITY ERROR« ausgegeben. Liegt Position hinter dem String-Ende, so wird ein String der Länge 0 als Ergebnis geliefert. Ist die Summe aus Position und Länge größer als die String-Länge, so werden nur Zeichen innerhalb des Strings übernommen.

Beispiel:

```
COPY('Beispielstring',1,8) liefert 'Beispiel'
COPY('Beispielstring',50,4) liefert ''
COPY('Beispielstring',9,99) liefert 'string'
```

LENGTH(String-Ausdruck)

Interne Repräsentation der Stringlänge

Es wird die Länge des String-Ausdruckes übergeben. Bei Strings wird die aktuelle Länge in dem Element mit dem Index 0 gespeichert:

```
ORD(S[0]) = LENGTH(S)
```

Diese Längenangabe können Sie auch bewußt manipulieren, sind jedoch durch das Laufzeitsystem dann nicht mehr vor Fehlern geschützt:

Beispiel:

```
PROGRAM STEST(INPUT,OUTPUT);
VAR S: STRING;
    I: INTEGER;
BEGIN
  FOR I:= 1 TO 20 DO S[I]:= CHR(I-1+ORD('A'));
  S[0]:= CHR(20);
  WRITELN(S);
END.
```

In diesem Beispiel wird ein String indiziert mit den ersten 20 Buchstaben des Alphabets gefüllt. Anschließend wird das Längenfeld des Strings auf die Länge von 20 Zeichen gesetzt. Übersichtlicher ist jedoch sicher die folgende Programmversion:

```
PROGRAM STEST2(INPUT,OUTPUT);
VAR S : STRING;
    CH: CHAR;
BEGIN
  S:= '';
  FOR CH:= 'A' TO 'O' DO S:= S + CH;
  WRITELN(S);
END.
```

POS(String-Ausdruck1,String-Ausdruck2)

Das Ergebnis dieser Funktion ist die Position des ersten Auftretens von String-Ausdruck1 in String-Ausdruck2. Ist String-Ausdruck1 nicht in String-Ausdruck2 enthalten, so ist das Funktionsergebnis die Zahl 0.

VAL(String-Ausdruck)

Umwandlungsfehler bei Zahlenkonvertierung

Mit dieser Funktion wird der Argument-String als eine Zahl interpretiert. Dabei werden Zahlen sowohl im Festkomma- als auch im Exponentialformat umgewandelt. Das Ergebnis der Funktion ist immer vom Typ REAL. Treten illegale Zeichen im String auf, so werden nur die vorausgehenden Ziffern interpretiert. Ist bereits das erste Zeichen illegal, so ist das Ergebnis der Umwandlung die Zahl 0.0.

Beispiele:

```
VAL('3.0') = 3.0
VAL('-3000.0E-7') = -3.0E-4
VAL('12XXX34') = 12.0
VAL('MCC') = 0.0
```

STR(Ausdruck)
STR(Ausdruck : N)
STR(Ausdruck : N : M)

Der Ausdruck vom Typ REAL wird wie bei der Prozedur WRITE beschrieben formatiert. Die Formatierungs-Parameter erlauben ebenfalls eine Definition der Länge des Ergebnis-Strings, die Wahl zwischen Festkomma- und Exponentialdarstellung sowie die Angabe der zu druckenden Nachkommastellen in beiden Formaten. Beispiele finden sich bei der Beschreibung der Prozedur WRITE.

4.4.4.8 Systemnahe vordefinierte Prozeduren

Vergleich Adressen mit INTEGER-Zahlen

Die in diesem Abschnitt besprochenen Funktionen und Prozeduren erwarten teilweise Adressen als Parameter. Da Adressen positive Werte zwischen 0 und 65535 annehmen, lassen sie sich nicht direkt als INTEGER-Zahlen angeben, die ja nur bis MAXINT=32767 reichen. Daher müssen Zahlen größer oder gleich 32768 durch die entsprechende negative Zahl im Zweierkomplement ersetzt werden:

Beispiel:

Die hexadezimale Zahl $FFE4 entspricht der Adresse 65508. Diese Zahl entspricht der INTEGER-Zahl –28 im Zweierkomplement, da 65508–256∗256 = –28 ist.

Um also aus einer Adresse X >= 32768 die zugehörige negative INTEGER-Zahl N zu erhalten, müssen Sie nur folgende Formel verwenden:

```
N = X - 256 * 256
```

POKE(Adresse,Wert)

Nachdem zuvor mit einer Zuweisung an die Variable BANK (siehe 4.4.4.3) die Speicherkonfiguration festgelegt wurde, wird durch die Prozedur POKE an der genannten Adresse der angegebene Wert vom Typ BYTE gespeichert. Die Adresse muß im Zweierkomplement angegeben werden.

Beispiel:

```
BANK:=0; POKE(1024,44)
```

setzt im Textmodus ein Komma in die linke obere Ecke des 40-Zeichen-Bildschirmes.

PEEK(Adresse)

Das Ergebnis dieser Funktion ist der Inhalt der angegebenen Adresse. Dieser Wert ist vom Typ BYTE. Die Speicherkonfiguration für diese Leseoperation kann zuvor durch die BANK-Variable definiert werden.

Beispiel:

```
BANK:=0; WRITE(PEEK(1024));
```

druckt den Code des Zeichens in der linken oberen Ecke des 40-Zeichen-Bildschirmes.

SYS(Adresse)
SYS(Adresse, Registervariable)

Registervariable für Maschinenprogramme

Mit diesem Befehl wird ein Unterprogramm in Maschinensprache an der angegebenen Adresse aufgerufen. Dieses Programm muß mit dem Befehl RTS enden. Das Programm darf den Prozessorstack sowie *heap* und *stack* des Pascal-Programmes nicht zerstören und die in Abschnitt 4.3.4 aufgezählten Zero-Page-Variablen (T bis TOP) nicht verändern.

bank-switching

Wie bei den Befehlen POKE und PEEK wird die Speicherkonfiguration bei Bedarf mit der Variablen BANK gewählt. Optional ist die Angabe einer Variablen, die die Werte der Prozessorregister vor und nach dem Aufruf enthält. Somit können in den Registern Parameter zwischen Pascal und Maschinensprache ausgetauscht werden. Die Register werden byteweise in der Reihenfolge Statusregister, Akkumulator, X- und Y-Register gespeichert.

Beispiel:

Die folgende Prozedur zeigt, wie man die LOAD-Routine des Betriebssystems aufruft. Zunächst werden mit den Prozeduren SETBNK, SETLFS und SETNAM die gewünschten Parameter gespeichert, mit denen anschließend die eigentliche LOAD-Routine ausgeführt wird. Abschließend wird überprüft, ob beim Laden ein Fehler auftrat, was durch ein gesetztes Carry angezeigt wird. In diesem Fall ist das Bit 0 im Statusregister gesetzt, somit hat das Statusregister einen ungeraden Wert.

```
PROCEDURE LOAD(S: STRING; TARGETBANK: INTEGER);
(* DIE PROZEDUR LÄDT DIE DATEI MIT DEM NAMEN S VON DER *)
(* DISKETTE IN DIE ANGEGEBENE BANK. DIE LADEADRESSE    *)
(* IST MIT DEM FILE GESPEICHERT WORDEN. DIE PROZEDUR   *)
(* ENTSPRICHT ALSO DEM BASIC-BEFEHL BLOAD"...",ONBx    *)
  VAR I   : INTEGER;
      REGS: RECORD
              SR,AKKU,X,Y: BYTE;
            END;
BEGIN
  BANK:=15;
  WITH REGS DO
    BEGIN
      AKKU:= TARGETBANK; X:=1;          (* NAME IN BANK 1 *)
      SYS(-152,REGS);                   (* SETBNK         *)
      AKKU:= LENGTH(S); I:= ADDU(ADR(S),1);
      X:= I; Y:= HBYTE(I);              (* ADRESSE NAME   *)
      SYS(-67, REGS);                   (* SETNAM         *)
                                        (* A = laden      *)
      AKKU:=0; X:=8; Y:=1;              (* X = Gerät      *)
                                        (* Y = absolut    *)
      SYS(-70,REGS);                    (* SETLFS         *)
      AKKU:=0; SYS(-43,REGS);           (* LOAD           *)
      IF ODD(SR) THEN                   (* CARRY=1 ?      *)
        WRITELN('LADEFEHLER!');
    END;
END; (* LOAD *)
```

ADDU(Ausdruck,Ausdruck)

Addition ohne Überlauftest

Das letztere Beispiel zeigt auch einen Anwendungsfall für die Funktion ADDU: Da in *Pascal 2.0 alle* arithmetischen Operationen (auch mit ganzen Zahlen) auf Überschreitung des Zahlenbereichs geprüft werden, kann es bei Rechnungen mit Adressen zu Bereichsüberschreitungen kommen. Würde im oberen Beispiel die Adresse des Strings S (ADR(S)) den Wert 32767 besitzen, so würde die Summation ADR(S)+1 zu einem Programmabbruch führen.
Bei der Funktion ADDU (*add unsigned*) werden jedoch die beiden INTEGER-Ausdrücke ohne Berücksichtigung von Überläufen addiert.

Beispiel:

```
ADDU(3,5)        =  8
ADDU(-1,-1)      = -2
ADDU(32767,1)    = -32768
ADDU(-32768,-1)  = 32767
```

HALT

Mit diesem Prozeduraufruf wird ein Pascal-Programm ohne Fehlermeldung sofort beendet.

FAST, SLOW

Umschaltung der Taktfrequenz des Prozessors

Mit diesen Prozeduren kann der Rechner vom 1-MHz-Modus in den 2-MHz-Modus und umgekehrt geschaltet werden. Im 2-MHz-Modus laufen alle Programme praktisch doppelt so schnell. Jedoch ist dann keine Anzeige auf dem 40-Zeichen-Bildschirm möglich.

HBYTE(Ausdruck)

Um das höherwertige Byte eines INTEGER-Wertes zu erhalten, kann diese Operation verwendet werden. Die Operation X DIV 256 liefert nur für X>=0 das gleiche Ergebnis.

SIZEOF(Typname)

In manchen Anwendungsfällen ist es sinnvoll, den Speicherplatzbedarf von Werten eines Typs in Bytes zu kennen. Diese Funktion SIZEOF liefert diese Größe vom Typ INTEGER.

Beispiel:

```
SIZEOF(TEXT) = 7
```

ADR(Variable)

Um die absolute Adresse einer Variablen in Bank 1 zu erhalten, kann man diese Funktion verwenden. Die Funktion kann auch auf ARRAY-Elemente, Record-Felder und dynamische Variablen angewandt werden. Diese Funktion wurde in diesem Abschnitt im Beispiel zur Prozedur SYS benutzt, um die Adresse des Strings S zu erhalten.

4.4.4.9 Vordefinierte Grafikprozeduren

Deklaration eines Grafikmodus im Programmkopf

In diesem Abschnitt werden die Funktionen und Prozeduren besprochen, die im Zusammenhang mit Bildschirmausgaben stehen. Einige Prozeduren arbeiten jedoch auch im Textmodus. Um Darstellungen in den Grafikmodi vorzunehmen, muß im Programmkopf der Name GRAPHIC stehen. Dadurch werden 8 Kbyte Speicherplatz für die *bit-map* reserviert. Fehlt dieser Speicherbereich für hochauflösende Grafik, so wird das Programm mit folgender Fehlermeldung abgebrochen:

```
NO GRAPHICS AREA ERROR
```

Parametersyntax der Grafikprozeduren

Die Namen und die Syntax der Prozeduren und Funktionen orientieren sich an den Vorgaben durch BASIC 7.0. Eine Implikation dieser Philosophie ist die Tatsache, daß viele Parameterlisten eine variable Länge besitzen oder daß nicht angegebene Parameter durch *default*-Werte (Standardvorgaben) ersetzt werden. Sie sollten sich der Tatsache bewußt sein, daß die vorgestellten Grafikroutinen Ihre Programme automatisch mit anderen Computern und Pascal-Compilern inkompatibel machen.

Bei der Beschreibung der Parameterlisten gelten folgende Konventionen:

1. Optionale Parameter werden in eckigen Klammern angegeben.
2. Parameter, die beliebig oft (0, 1, 2,.. mal) wiederholt werden können, werden in geschweiften Klammern genannt.
3. Bei Auslassung von Parametern am Ende der Parameterliste müssen keine Kommata für fehlende Parameter angegeben werden. Die Liste kann also direkt mit einer Klammer zu beendet werden.

Beispiel für default-Parameter

Beispiel:

Wegen Regel 1 kann GRAPHIC(4,0,19) abgekürzt werden zu GRAPHIC(4,,). Diese Parameterliste kann wegen Regel 3 zu GRAPHIC(4) verkürzt werden. Regel 2 wird z.B. bei der DRAW-Anweisung angewendet.

Koordinaten

Koordinaten sind Integer-Zahlenpaare in der Reihenfolge X- und dann Y-Koordinate. Die darstellbaren Koordinaten für X und Y richten sich nach der momentan gewählten Auflösung (siehe GRAPHIC).

Standard-Zeichenfarbe

Wird bei einer Grafikoperation keine Farbe angegeben, so wird die Farbe 1 (aktuelle Vordergrundfarbe) verwendet.

Pixelcursor und default-Koordinaten

Jede Zeichenoperation setzt außerdem den sogenannten Pixelcursor. Dies ist die Koordinate des letzten gezeichneten Punktes auf dem Bildschirm. Kann bei einer Grafikoperation eine Koordinate entfallen, so wird eine nicht angegebene Koordinate durch die Position des Pixelcursors ersetzt.

Beispiel:

```
DRAW(1,0,0,50,50)
```

Nach dieser Operation steht der Pixelcursor auf dem Endpunkt der Strecke mit den Koordinaten 50,50. Erfolgt anschließend die Operation

```
DRAW(1,,0,50)
```

so wird die fehlende Ausgangskoordinate durch die Position des Pixelcursors (50,50) ersetzt.

GRAPHIC(Modus,[Löschflag],[Textzeile])

Grafikmodi und ihre Auflösung

Mit diesem Befehl schalten Sie zwischen den verschiedenen Grafik- und Textmodi hin und her. Folgende Modi werden unterstützt:

0 Textdarstellung auf dem 40-Zeichen-Bildschirm.

1 Hochauflösende Grafik. Anzeigbare X-Koordinaten liegen zwischen 0 und 319. Anzeigbare Y-Koordinaten liegen zwischen 0 und 199. Es sind nur die Farben 0 und 1 zulässig.

2 Grafikmodus wie 1. Zusätzlich können am unteren Bildschirmrand Textzeilen des 40-Zeichen-Bildschirms eingeblendet werden.

3 Mehrfarbengrafik. Anzeigbare X-Koordinaten liegen zwischen 0 und 159. Anzeigbare Y-Koordinaten liegen zwischen 0 und 199. Es können die Farben 0, 1, 2 und 3 gewählt werden.

4 Grafikmodus wie 3. Zusätzlich können am unteren Bildschirmrand Textzeilen des 40-Zeichen-Bildschirms eingeblendet werden.

5 Textdarstellung auf dem 80-Zeichen-Bildschirm.

Das Lösch-Flag darf nur die Werte 0 und 1 annehmen. Bei der Angabe des Wertes 1 wird der Bildschirm beim Einschalten der Modi 1 bis 4 gelöscht. Voreingestellt ist der Wert 0.

In den Modi 2 und 4 kann die erste darzustellende Textzeile im Bereich 0 bis 24 gewählt werden. Voreingestellt ist der Wert 19.

PAINT([Farbe],[Koordinaten],[Modus])

Ausmalen einer Fläche

Mit diesem Befehl wird ein umrandeter Bildschirmbereich, ausgehend von den genannten Koordinaten, mit einer Farbe ausgemalt. Mit dem Modus wird bestimmt, ob die Grenze der auszumalenden Figur genau die genannte Farbe besitzen muß (0) oder ob die Grenze durch jede Farbe außer der Hintergrundfarbe gebildet wird (1). Voreingestellt ist der Modus 0. Der Startpunkt muß innerhalb der auszumalenden Figur liegen.

Während der Ausführung des Befehls ist die Run/Stop-Taste gesperrt, da während des Ausmalvorganges kein definiertes Beenden des Pascal-Programmes zu realisieren ist.

DISPLAY([Farbe],Koordinaten,String-Ausdruck,[Inversflag])

DISPLAY im Textmodus

In BASIC heißt diese Prozedur CHAR. Es wird das Ergebnis des String-Ausdruckes am Bildschirm ab den angegebenen Anfangskoordinaten in der gewählten Farbe ausgegeben. Diese Prozedur kann auch im *Textmodus* 0 und 5 verwendet werden. Besitzt das Inversflag den Wert 1, so wird der Text invertiert (Vordergrund und Hintergrund vertauscht) ausgegeben.

Steuerzeichen (z.B. zur Cursor-Bewegung) sind nur im Textmodus erlaubt. Die X- und Y-Koordinaten werden von 0 beginnend gezählt und beziehen sich im Textmodus relativ zum momentan aktiven Window. Um im Mehrfarbenmodus Zeichen in der Zusatzfarbe 2 anzuzeigen, müssen die Farbe 0 und das Inversflag = 1 gewählt werden.

Beispiel:

```
DISPLAY(1,20,0,'UEBERSCHRIFT IN ZEILE 0',1)
```

BOX([Farbe],Koordinaten,[Koordinaten],[Winkel],[Ausmalflag])

Rechtecke

Mit dieser Prozedur können Sie Rechtecke frei wählbarer Orientierung am Bildschirm darstellen. Die Koordinaten definieren die linke obere und die rechte untere Ecke des Rechteckes, wobei die Seiten des Rechteckes parallel zur X- und Y-Achse liegen. Durch die Angabe eines Winkels in Grad (ungleich 0) wird das so definierte Rechteck im Uhrzeigersinn gedreht. Schließlich können Sie noch durch Setzen des Ausmalflags das Rechteck in der gewählten Farbe füllen. Voreinstellung für Winkel und Ausmalflag ist jeweils 0.

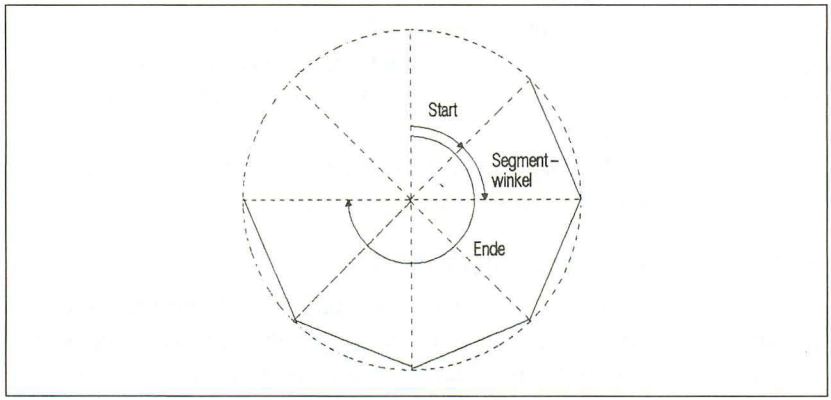


Bild 119: Parameter der Prozedur CIRCLE

CIRCLE([Farbe],[Koordinaten],XRadius,[YRadius],[Start],[Ende],
[Rotationswinkel],[Segmentwinkel])

Kreise und Polygon

Der Name dieser Prozedur verbirgt etwas die zahlreichen Möglichkeiten, die in diesem Befehl stecken. Am besten betrachten Sie Bild 119 zur Erläuterung der einzelnen Parameter. Durch die Mittelpunktskoordinaten, den X- und Y-Radius wird eine Ellipse definiert (gestrichelt gezeichnet). Nun werden zwischen dem Winkel »Start« und dem Winkel »Ende« Stützstellen auf der Ellipse berechnet. Diese Stützstellen haben einen Abstand von »Segmentwinkel« Grad. Durch die Verbindung der Stützstellung mit Strecken entsteht ein Polygonzug. Wird ein kleiner Segmentwinkel gewählt, nähert sich die Figur immer mehr der gestrichelten Ellipse an.

Schließlich läßt sich noch die gesamte Figur um den Rotationswinkel im Uhrzeigersinn drehen. Die folgende Tabelle gibt Aufschluß über die zulässigen Werte der Parameter und die Standardvorgaben, falls ein Parameter nicht spezifiziert wird:

Parameter	Wertebereich	Standardvorgabe
Farbe	0..3	1
Koordinaten	s. Einführung	Pixel-Cursor
XRadius	–500..500	Größere Werte sind zwar möglich, werden jedoch nicht mehr sinnvoll dargestellt.
YRadius	–500..500	Je nach Auflösung und XR, so daß ein Kreis entsteht
Start	0..32767	0
Ende	0..32767	360
Rotationswinkel	0..32767	0
Segmentwinkel	1..255	2

Beispiele:

```
CIRCLE(,160,100,65,10)                          (Ellipse)
CIRCLE(,160,100,60)                               (Kreis)
CIRCLE(,160,100,60,,,,45)                       (Achteck)
CIRCLE(,160,100,60,10,0,90)                (Ellipsensegment)
CIRCLE(,160,100,60,10,,,45,12)   (um 12 Grad gedrehtes Achteck)
```

DRAW([Farbe],[Koordinaten],Koordinaten {,Koordinaten})

Linien, Punkte und Polygone

Mit diesem Befehl zeichnen Sie Linien und Punkte in der gewählten Farbe. Werden die Startkoordinaten nicht genannt, so wird die Position des Pixel-Cursors als Ausgangspunkt verwendet. Durch die Angabe von mehr als einem Zielkoordinatenpaar wird ein Polygonzug zwischen den Koordinaten gezogen. Nach der Ausführung des Befehls steht der Pixel-Cursor auf dem letzten gemalten Punkt.

LOCATE(Koordinaten)

Pixel-Cursor setzen

Um die Position des Pixel-Cursors gezielt zu setzen, können Sie die Prozedur LOCATE verwenden. Durch den Aufruf der Prozedur wird keine Ausgabe auf dem Bildschirm erzeugt. Dieser Prozeduraufruf hat keinen Einfluß auf den Text-Cursor.

Beispiel:

```
LOCATE(160,100);CIRCLE(1,,60)
```

Mit dieser Anweisungsfolge wird zunächst der Pixel-Cursor (unsichtbar) auf die Bildschirmmitte gesetzt. Anschließend wird ein Kreis mit dem Radius 60 um diesen Punkt gezeichnet.

```
DISPLAY(1,0,19,'')
```

Dieser Befehl setzt den Text-Cursor in Zeile 20 und Spalte 1. Als Textfarbe wird die Farbe Schwarz gewählt.

COLOR(Farbe, Farbcode)

Farbattribute

Bei den verschiedenen Auflösungsmodi steht eine unterschiedliche Anzahl an Farben zur Verfügung. Durch die Speicherung der Farbattribute des Bildschirms können Sie zwar jede beliebige Farbe als Zeichenfarbe wählen, jedoch kann durch das Setzen eines Punktes in dieser Farbe ein Teil der in der »Umgebung« des Punktes gesetzten Pixel mit eingefärbt werden. Im hochauflösenden Grafikmodus kann ein Punkt nur die Hintergrund- oder Vordergrundfarbe besitzen. Zu diesen »zwei« Farben treten im Mehrfarbenmodus noch zwei Zusatzfarben.

Mit dem Befehl COLOR können Sie diesen »Farben« (Hintergrund, Vordergrund, Zusatzfarbe 1 und 2) Farbcodes zuordnen, die folgende Werte annehmen können:

1	schwarz	9	hellbraun
2	weiß	10	braun
3	rot	11	rosa
4	grün	12	dunkelgrau
5	violett	13	grau
6	dunkelgrün	14	hellgrün
7	blau	15	hellblau
8	gelb	16	hellgrau

Die Farben werden ebenfalls codiert angegeben:

0	Hintergrund (40-Zeichen-Modus)
1	Zeichenfarbe (hochauflösende Grafik / Mehrfarbenmodus)
2	Zusatzfarbe 1 (nur im Mehrfarbenmodus)
3	Zusatzfarbe 2 (nur im Mehrfarbenmodus)
4	Randfarbe (40-Zeichen-Modus)
5	Textfarbe (40- und 80-Zeichen-Modus, für WRITE-Befehle)
6	Hintergrundfarbe (80-Zeichen-Modus)

Beispiel:

```
COLOR(0,2); COLOR(1,3); COLOR(4,1); COLOR(5,7)
```

wählt die folgenden Farben: Hintergrundfarbe weiß, Zeichenfarbe rot, Rand schwarz und Textfarbe blau.

SCNCLR
SCNCLR(Modus)

Bildschirme löschen

Ohne Parameterangabe wird der Bildschirm des momentan aktiven Grafikmodus gelöscht. Durch die Angabe eines Parameters kann der zu löschende Bildschirm explizit gewählt werden:

0	Der 40-Zeichen-Text-Bildschirm
1	Der hochauflösende Grafikbildschirm
2	Der hochauflösende Grafikbildschirm mit Textzeilen
3	Der Mehrfarben-Grafikbildschirm
4	Der Mehrfarben-Grafikbildschirm mit Textzeilen
5	Der 80-Zeichen-Bildschirm
–1	Der aktuelle Bildschirm wird gelöscht (entspricht SCNCLR ohne Parameter).

Beim Löschen eines Grafikbildschirms werden alle Pixel auf dem Bildschirm in der Hintergrundfarbe gesetzt.

WIDTH(Breite)

Linienbreite

Bei allen Zeichenbefehlen (DRAW, CIRCLE, BOX) kann die Strichstärke verändert werden. Voreingestellt ist die Breite 1. Alternativ kann die Breite 2 gewählt werden, wodurch immer zwei nebeneinanderliegende Punkte gesetzt werden. Andere Werte für die Breite führen zur Fehlermeldung »ILLEGAL QUANTITY ERROR«.

RGR

Aktuellen Grafikmodus abfragen

Mit dieser parameterlosen Funktion können Sie den eingestellten Grafikmodus bestimmen. Der gelieferte Wert vom Typ INTEGER liegt zwischen 0 und 5 entsprechend der Tabelle für die Modi bei der Prozedur GRAPHIC.

RCLR(Farbe)

Aktuelle Farbwahl abfragen

Als Funktionsergebnis erhalten Sie den aktuellen Farbcode (1 bis 16) vom Typ INTEGER für diese Farbe. Gültige Farben liegen im Bereich von 0 bis 6. Die Farben sind wie bei der Prozedur COLOR codiert.

Beispiel:

```
ZEICHENFARBE:= RCLR(1)
```

Der Variablen ZEICHENFARBE wird der Farbcode der Zeichenfarbe im Grafikmodus zugewiesen. Wurde z.B. die Zeichenfarbe Rot gewählt, so besitzt anschließend die Variable ZEICHENFARBE den Wert 3.

RDOT(Code)

Aktuelle Pixel-Cursor-Position abfragen

Mit dieser Funktion erhalten Sie Informationen über die Position des Pixel-Cursors. Der Code darf folgende Werte annehmen:

Farbe eines Punktes abfragen

RDOT(0)	liefert die X-Koordinate des Pixel-Cursors
RDOT(1)	liefert die Y-Koordinate des Pixel-Cursors
RDOT(2)	liefert die Farbe an der Position des Pixel-Cursors

Beispiel:

```
LOCATE;(50,50); IF RDOT(2)<>0 THEN
WRITELN('Bei (50,50) ist ein Punkt gesetzt');
```

4.4.4.10 Weitere vordefinierte Prozeduren

WINDOW(XLO,YLO,XRU,YRU,[LÖSCHFLAG])

Bildschirmfensterdefinition

Es wird ein Bildschirmfenster für nachfolgende Textausgaben mit WRITE oder DISPLAY durch die Angabe der Koordinaten der linken oberen und rechten unteren Ecke definiert. Gültige Koordinaten in X-Richtung liegen im Bereich von 0 bis 79 (oder 39), während gültige Y-Koordinaten (Zeilennummern) von 0 bis 24 reichen. Dabei muß gelten:

XLO < XRU und YLO < YRU

Wird für das Löschflag der Wert 1 angegeben, so wird das Fenster nach der Definition mit der Hintergrundfarbe gelöscht. Um eine Window-Definition rückgängig zu machen, genügt die Steuerzeichenfolge <HOME> <HOME>.

Beispiel:

```
PROGRAM WINDOWS(INPUT, OUTPUT);
  CONST RESETWINDOW = #19 #19;
  VAR I: INTEGER;
BEGIN
  SCNCLR;
  WINDOW(20,5,40,8);
     FOR I:= 1 TO 18 DO WRITE('ZEILE',I);
  WRITE(RESETWINDOW, 'OHNE WINDOW');
END.
```

RWINDOW(Code)

Aktuelles Bildschirmfenster abfragen

Um die Größe des momentan aktiven Windows zu bestimmen, können Sie die Funktion RWINDOW verwenden. Der Code liegt im Bereich von 0 bis 2 und hat folgende Bedeutung:

RWINDOW(0)	liefert die Nummer der letzten Zeile im aktuellen Window (0 bis 24).
RWINDOW(1)	liefert die Nummer der letzten Spalte im aktuellen Window (0 bis 79).
RWINDOW(2)	liefert die maximale *Bildschirmbreite* (40 oder 80 Zeichen).

SOUND(Stimme,Frequenz,Dauer,[Richtung],[Maxfreq],[Stufe],[Welle],[Impulsbreite])

Ton- und Geräuscherzeugung

Mit diesem Prozeduraufruf wird ein Geräuscheffekt gestartet, der eine programmierbare Zeit andauert. Während dieser Zeit können andere Anweisungen ausgeführt werden. Es können drei Stimmen unabhängig voneinander programmiert werden. Bei aufeinanderfolgenden SOUND-Anweisungen für die gleiche Stimme wird so lange gewartet, bis die vorhergehende Anweisung vollständig ausgeführt wurde.
Als Stimme können die Zahlen 1, 2 und 3 gewählt werden. Die Frequenz kann jeden beliebigen INTEGER-Wert annehmen. Dabei werden negative Werte als Frequenzen größer als 32767 interpretiert. Die Dauer einer Tonausgabe wird in Einheiten von $^1/_{60}$ Sekunde definiert. Wird als Dauer Null angegeben, so werden alle laufenden Klangeffekte für diese Stimme sofort abgebrochen.
Durch die Angabe einer Richtung können auch zeitlich veränderliche Tonausgaben erzeugt werden:

0	zunehmende Tonhöhe (Voreinstellung)
1	abnehmende Tonhöhe
2	oszillierende Tonhöhe

Die Maximalfrequenz beschränkt ansteigende Klangeffekte nach oben hin. Die Voreinstellung beträgt 0. Der Parameter Welle darf nur folgende Werte annehmen, die die Wellenform für die Stimme festlegen:

0	Dreieck
1	Sägezahn
2	Rechteck (Voreinstellung)
3	Rauschen

Nur im Falle einer Rechteckwelle (Welle = 2) wird der letzte Parameter, die Impulsbreite, ausgewertet. Er darf Werte zwischen 0 und 4095 annehmen. Dabei bedeutet z.B. der voreingestellte Wert von 2048, daß die positive und negative Halbwelle gleich breit sind.

VOL(Lautstärke)

Lautstärke

Dieser Prozeduraufruf legt die Lautstärke für alle nachfolgend mit SOUND-

und PLAY-Prozeduren erzeugten Töne fest. Gültige Werte liegen zwischen 0 und 15, wobei die Ausgabe bei einem Wert von 0 ausgeschaltet ist.

FILTER(Frequenz,[Tiefpaßflag],[Bandpaßflag],[Hochpaßflag], [Resonanz])

Filter

Den tonerzeugenden Generatoren (Stimme 1 bis 3) ist ein Filter nachgeschaltet, mit dem sich die Frequenzverteilung des Signals modifizieren läßt. Jeder der drei Filter läßt sich unabhängig von den anderen über ein Flag ein- (1) und ausschalten (0). Wird das Flag auf –1 gesetzt oder der Parameter in der Parameterliste ausgelassen, so bleibt der Filter unverändert. Der Parameter Resonanz gibt – grob gesprochen – die Selektivität der Filter an. Dabei sind Werte zwischen 0 und 15 erlaubt. Wird ein negativer Wert angegeben oder die Resonanz in der Parameterliste nicht aufgeführt, so bleibt eine zuvor eingestellte Resonanz erhalten.

Beispiel:

```
FILTER(1200,1,,0,10)
```

Mit diesem Befehl wird die Filterfrequenz auf 1200 festgelegt, der Tiefpaß wird eingeschaltet, der Bandpaß bleibt unverändert, während der Hochpaß ausgeschaltet wird. Die Resonanz wird auf ein gut wahrnehmbares Niveau eingestellt.

PLAY(String-Ausdruck)

Notenfolge spielen

Mit dieser Prozedur können Musikstücke (nach Noten) programmiert werden. Die Programmausführung wird (im Unterschied zum SOUND-Befehl) erst fortgesetzt, wenn der gesamte String-Ausdruck abgearbeitet wurde. Innerhalb des Strings sind neben den eigentlichen Notennamen (C D E F G A B) zahlreiche andere Zeichen zulässig, mit denen man Stimme, Pausen, Tondauer, Oktave, Lautstärke und Filter wählen kann. Jede der drei Stimmen ist separat programmierbar. Nähere Einzelheiten über die Syntax der »Musik-Strings« finden Sie im BASIC-Handbuch.

Beispiel:

```
PLAY('QCDEFHGGQAAAG')
```

TEMPO(Geschwindigkeit)

Während innerhalb des »Musik-Strings« bei der Prozedur PLAY die Tondauer in ganzen, halben, viertel, achtel oder sechzehntel Tonlängen gewählt wird, bestimmt die Prozedur TEMPO die absolute Abspielgeschwindigkeit des Strings. Die Geschwindigkeit liegt im Bereich von 0 bis 255. Hohe Werte führen zu einer größeren Geschwindigkeit. Im BASIC-Handbuch wird folgende Formel zur Berechnung der Dauer einer ganzen Note genannt:
(19.22 / Geschwindigkeit) Sekunden

ENVELOPE(Nummer,[Anschlagzeit],[Abschwellzeit],[Haltezeit], [Ausklingzeit],[Wellenform],[Impulsbreite])

Hüllkurvendefinition

Für PLAY-Anweisungen existieren 10 verschiedene »Musikinstrumente«. Die unterschiedlichen Klangfarben dieser Instrumente werden durch die Wahl der Hüllkurven erzeugt. Um jeweils eine dieser Hüllkurven zu verändern, also ein neues »Instrument« zu definieren, können Sie die ENVELOPE-Prozedur verwenden. Zunächst wählen Sie die Nummer des »Instrumentes« (0 bis 9), das Sie verändern wollen. In der Parameterliste müssen Sie nur Parameter nennen, die Sie verändern wollen. Negative bzw. ausgelassene Parameter werden nämlich durch die zuvor gültigen Werte der Hüllkurven ersetzt.
Die Anschlag-, Abschwell-, Halte- und Ausklingzeiten dürfen Werte zwischen 0 (kurz) und 15 (lang) annehmen. Die Wellenform ist eine Zahl zwischen 0 und 4:

0	Dreieck
1	Sägezahn
2	Rechteckwelle
3	Rauschen
4	Ringmodulation

Wie bei der SOUND-Prozedur wird der Parameter Impulsbreite (aus dem Intervall 0 bis 4095) nur für Rechteckwellen ausgewertet.

Beispiel:

```
ENVELOPE(0,4,4,8,5,2,2048)
```

Mit diesem Prozeduraufruf wird die Hüllkurve 0 (»Klavier«) neu definiert. Tritt in einem »Musik-String« bei der Prozedur PLAY der Befehl »T0« auf, so wird die mit ENVELOPE definierte Hüllkurve verwendet.

POT(Nummer)

Drehreglerabfrage

Diese Funktion liefert die Position eines der vier anschließbaren *paddles* (Drehregler). Der Parameter »Nummer« vom Typ INTEGER muß dementsprechend zwischen 1 und 4 liegen. Je nach der Stellung des Drehreglers wird ein Wert zwischen 0 und 255 geliefert. Wird bei dem gewählten Regler auch der Feuerknopf betätigt, so wird zu diesem Wert noch 256 addiert.

Beispiel:

```
P:= POT(3);
WRITELN('STELLUNG REGLER 3 =',P AND 255);
IF P>255 THEN WRITELN('FEUERKNOPF GEDRUECKT');
```

PEN(Code)

Lichtstiftabfrage

Mit dieser Funktion wird der Lichtstift (*light pen*) abgefragt. Mit dem Code (Typ INTEGER) wird der Abfragemodus definiert:

0	X-Koordinate des *light pen* auf dem Grafikbildschirm (60 bis 320)
1	Y-Koordinate des light pen auf dem Grafikbildschirm (50 bis 200)
2	Text-Spalte, in der sich der *light pen* auf dem 80-Zeichen-Bildschirm befindet
3	Text-Zeile, in der sich der *light pen* auf dem 80-Zeichen-Bildschirm befindet
4	Es wird eine 1 übergeben, falls der *light pen* seit der letzten Abfrage (mit PEN) aktiviert wurde, andernfalls 0.

Beispiel:

```
REPEAT UNTIL PEN(4)<>0;
WRITELN('STIFT IST BEI',PEN(2),PEN(3), 'AKTIVIERT');
```

JOY(Nummer)

Joystick-abfrage

Diese Funktion liefert codiert die Position eines der beiden Joysticks. Die Joysticks besitzen die Nummern 1 und 2. Folgende Werte können auftreten:

0	Mittelstellung
1	nach vorne
2	nach vorne rechts
3	nach rechts
4	nach hinten rechts
5	nach hinten
6	nach hinten links
7	nach links
8	nach vorne links

Zu diesen Werten wird die Zahl 128 addiert, falls zusätzlich der Feuerknopf betätigt wurde.

4.4.5 Files in Pascal 2.0

Zur Anpassung an das »Betriebssystem« des C 128 mußte die Behandlung von Files in einigen Details gegenüber den Vorgaben des report geändert werden. Diese Unterschiede wurden bereits in Kapitel 2 und bei der Vorstellung der File-Prozeduren genannt. In diesem Abschnitt wird ein zusammenfassender Überblick über das *file-handling* in *Pascal 2.0* mit direkten Vergleichen zu anderen Implementierungen gegeben.

Externe und interne Dateien im report

Im *report* existiert eine Unterscheidung zwischen externen und internen Dateien. Externe Dateien existieren eventuell bereits vor dem Beginn und auch nach dem Ende eines Programmes. Die statischen File-Variablen für diese Dateien werden deshalb als Programmparameter im Programmkopf genannt. In *Pascal 2.0* dürfen dort nur die Standard-Files INPUT und OUTPUT auf-

geführt werden, von denen zumindest das File OUTPUT deklariert werden muß. Alle anderen Files sind gemäß den Konventionen des *report* interne Dateien, das heißt, sie existieren nur während des Programmlaufes, und andere Programme können nicht auf sie zugreifen.
Wie in den meisten Implementierungen für Mikrocomputer sind alle Dateien in *Pascal 2.0* gemäß der obigen Terminologie externe Dateien: Sie werden auf Peripheriegeräten unter einem File-Namen gespeichert und bleiben auch nach dem Programmende erhalten. Eine Konsequenz dieser Speicherungsform ist, daß die Prozeduren RESET und REWRITE zusätzliche Parameter benötigen, um das Peripheriegerät und den File-Namen zu spezifizieren.

RESET und REWRITE ersetzt durch OPEN und CLOSE

Aus Gründen der Klarheit und Kompatibilität mit BASIC-Programmen werden statt RESET und REWRITE die Prozeduren OPEN und CLOSE verwendet. Dabei müssen Sie beachten, daß die im *report* definierte Prozedur RESET bereits den ersten Satz der Datei in die Puffer-Variable übernimmt, das heißt:

```
REWRITE(F)   entspricht OPEN(F,8,Sekundäradresse,Filename + 'S,W');
RESET(F)     entspricht OPEN(F,8,Sekundäradresse,Filename + 'S,R');
             und anschließend  GET(F)
```

Jedes File F wird intern mittels eines eigenen Deskriptors verwaltet, der neben der Puffervariablen (F↑) Informationen über EOF(F), STATUS(F) und EOLN(F) enthält, die bei jeder Ein- und Ausgabeoperation mit dieser File-Variablen aktualisiert werden. Deshalb können Files natürlich Bestandteil anderer Datenstrukturen(Array, Record, aber *nicht* File) sein und auch in Prozeduren und Funktionen *rekursiv* definiert werden.
Files, die zum Schreiben oder auch nur zum Lesen eröffnet waren, müssen mit CLOSE wieder geschlossen werden.

GET und PUT im Vergleich zu READ und WRITE

Hier soll noch einmal daran erinnert werden, daß GET und PUT die Daten in binärcodierter Form (gemäß der in Abschnitt 4.4.4.2 beschriebenen internen Speicherverteilung) lesen und schreiben, so daß diese Files nicht direkt ausgedruckt oder von BASIC-Programmen verwendet werden können. Falls ein Ausdruck oder eine Weiterverarbeitung erwünscht ist, können die Prozeduren READ(LN) und WRITE(LN) verwendet werden. Natürlich sind mit GET und PUT erzeugte Dateien wesentlich kompakter und schneller in der Bearbeitung, da keinerlei Konvertierung der Datenformate erforderlich ist.

Die Puffervariable

Eine weitere Abweichung vom *report* besteht darin, daß die Puffer-Variable (F↑) beim Lesen von einem File F nicht das nächste, sondern das *zuletzt* gelesene Zeichen enthält. Dies ist natürlich ein gravierender Nachteil, da damit keine Möglichkeit besteht, über die Variable F↑ ein Zeichen »vorauszuschauen« (*look ahead*). Der Grund für diese Änderung besteht darin, daß bei der Tastatureingabe in Dialogprogrammen ein »Vorausschauen« nicht sinnvoll zu realisieren ist.

Die Funktion STATUS(F) ermöglicht eine Kontrolle über die Ergebnisse von Ein- und Ausgabeoperationen. Ausdrücklich sei darauf hingewiesen, daß STATUS(F) (anders als die Variable ST in BASIC) nicht durch Ein- oder Ausgaben auf einem anderen File als F beeinflußt werden kann.

Umleitung der Standard-eingabe und -ausgabe

Die Standard-Files INPUT und OUTPUT sind zwar zu Programmbeginn den Geräten Tastatur und Bildschirm zugeordnet, jedoch läßt sich diese Zuordnung wie bei allen anderen Files während des Programmablaufes dynamisch mit der Prozedur OPEN ändern:

Beispiel:

Um alle nachfolgenden Bildschirmausgaben am Drucker (Geräteadresse 4, Sekundäradresse 0) zu protokollieren, eignet sich der Prozeduraufruf:

```
OPEN(OUTPUT,4,0)
```

Andererseits lassen sich Eingaben, die sonst von der Tastatur erfolgen, auch von einem beliebigen Gerät lesen. Mit den folgenden Parametern werden Eingaben vom File »COMMAND.BAT« gelesen:

```
OPEN(INPUT,8,3,'COMMAND.BAT,S,R')
```

Diese Umleitungen der Aus- und Eingabe werden mit den Befehlen CLOSE(INPUT) und CLOSE(OUTPUT) rückgängig gemacht.

4.4.5.1 Übernahme von Programmen mit Files

Im vorausgegangenen Abschnitt wurden die File-Prozeduren in *Pascal 2.0* mit ihren im *report* definierten Gegenstücken verglichen, so daß Sie Programme in »Standard-Pascal« durch kleine Änderungen an die Eigenarten des C128 anpassen können.

Vergleich Pascal 2.0 mit Turbo-Pascal

In Zeitschriften und Pascal-Büchern werden Sie jedoch häufiger auf Programme in *Turbo-Pascal* stoßen. Diese Implementation verwendet bei File-Operationen Routinen der Betriebssysteme CP/M bzw. MSDOS und PCDOS und weicht deshalb seinerseits vom *report* ab. In *Turbo-Pascal* existieren ebenfalls keine Unterschiede zwischen externen und internen Files, so daß diese auch nicht im Programmkopf aufgeführt werden. Der File-Name einer Datei muß vor dem ersten Aufruf der Prozeduren RESET und REWRITE mit ASSIGN festgelegt werden. Außerdem müssen Dateien nach der Bearbeitung mit CLOSE geschlossen werden.

In *Turbo-Pascal* existiert bei einem File F keine Puffervariable F↑. Daher werden die Prozeduren PUT und GET durch die Prozeduren READ und WRITE ersetzt. Dabei gilt:

```
READ(F,X)          entspricht     X:= F↑: GET(F);
WRITE(F,X)         entspricht     F↑:= X; PUT(F);
```

Beispiel:

Die beiden folgenden Programme zeigen das Erstellen und Lesen einer sequentiellen Datei in Turbo-Pascal und Pascal 2.0.

```
PROGRAM FILEDEMO(INPUT,OUTPUT);
(* SCHREIBE / LESE LEERE PRODUKTDATEI IN TURBO-PASCAL *)
      TYPE PRODUCTRECORD =
          RECORD
            NUMBER : INTEGER;
            NAME   : STRING[30];
            INSTOCK: INTEGER;
          END;
  VAR  PRODUCT     : INTEGER;
       P           : PRODUCTRECORD;
       PRODUCTFILE : FILE OF PRODUCTRECORD;

BEGIN
  ASSIGN(PRODUCTFILE,'PFILE');
  REWRITE(PRODUCTFILE);
  P.NAME:= ''; P.INSTOCK:= 0;
  FOR PRODUCT:= 1 TO 50 DO
     BEGIN
       P.NUMBER:= PRODUCT; WRITE(PRODUCTFILE,P);
     END;
  CLOSE(PRODUCTFILE);

  RESET(PRDUCTFILE);
  WHILE NOT EOF(PRODUCTFILE) DO
    BEGIN
      READ(PRODUCTFILE,P);
      WITH P DO
        WRITELN(NUMBER:5,NAME:20,INSTOCK:8);
    END;
  CLOSE(PRODUCTFILE);
END.

PROGRAM FILEDEMO(INPUT,OUTPUT);
(* SCHREIBE / LESE LEERE PRODUKTDATEI IN PASCAL 2.0 *)
  TYPE PRODUCTRECORD =
          RECORD
            NUMBER : INTEGER;
            NAME   : STRING[30];
            INSTOCK: INTEGER;
          END;
```

```
  VAR  PRODUCT     : INTEGER;
       PRODUCTFILE : FILE OF PRODUCTRECORD;

BEGIN
  OPEN(PRODUCTFILE,8,3,'PFILE,SEQ,WRITE');
  PRODUCTFILE↑.NAME:= ''; PRODUCTFILE↑.INSTOCK:= 0;
  FOR PRODUCT:= 1 TO 50 DO
     BEGIN
       PRODUCTFILE↑.NUMBER:= PRODUCT;
       PUT(PRODUCTFILE);
     END;
  CLOSE(PRODUCTFILE);

  OPEN(PRDUCTFILE,8,3,'PFILE,SEQ,READ');
  GET(PRODUCTFILE);
  WHILE NOT EOF(PRODUCTFILE) DO
    BEGIN
      WITH PRODUCTFILE↑ DO
         WRITELN(NUMBER:5,NAME:20,INSTOCK:8);
      GET(PRODUCTFILE);
    END;
  CLOSE(PRODUCTFILE);
END.
```

In Pascal 2.0 kann also die Variable P entfallen, da die auszugebenden und einzulesenden Daten direkt in der Puffervariablen PRODUCTFILE↑ vorliegen. Außerdem wurden die ASSIGN-, RESET- und REWRITE-Prozeduren durch OPEN-Prozeduren ersetzt. Durch die Wahl der Geräteadresse, des File-Typs und des File-Namens in *Pascal 2.0* werden die Daten ebenfalls unter dem Namen »PFILE« auf Diskette gespeichert.

GET nach OPEN erforderlich

Schließlich sollten Sie noch beachten, daß direkt nach dem Eröffnen der Datei PRODUCTFILE zum Lesen die Operation GET(PRODUCTFILE) ausgeführt wird, damit die Puffervariable bereits den ersten Record des Files enthält.

4.4.6 Aktive Kommentare

Kommentare innerhalb eines Pascal-Programmes dürfen beliebige Zeichenfolgen enthalten. In *Pascal 2.0* haben Kommentare, die direkt nach der öffnenden Kommentarklammer mit einem Dollarzeichen beginnen, eine Steuerfunktion für den Übersetzungsvorgang:

Syntaxaktive Kommentare

```
(* $ Dies ist ein normaler Kommentar *)
(*$R+ Dieser Kommentar schaltet die Bereichsüberprüfung ein *)
```

Ein aktiver Kommentar kann auch mehrere Optionen ein- und ausschalten. Dazu müssen die Schlüsselbuchstaben durch Kommata (ohne Leerstellen) getrennt aufgeführt werden:

```
(*$R+,C-,P+  range-check ein,  post-mortem-chain  aus,
parametercheck ein *)
```

Die Wirkung eines aktiven Kommentars gilt für alle nachfolgenden Quelltextzeilen (auch innerhalb von Include-Files), bis ein gegenteiliger aktiver Kommentar gelesen wird.

4.4.6.1 Bereichsüberprüfung

Bei String-Zuweisungen, Parameterübergabe von Strings und allen Parametern der Standardprozeduren wird *immer* eine Bereichsüberprüfung auf gültige Werte vorgenommen. Auch die Ergebnisse arithmetischer Operationen mit reellen und ganzen Zahlen werden *immer* auf Überschreitung des darstellbaren Zahlenbereiches geprüft.

Der Compiler besitzt einen *Schalter*, mit dem die Erzeugung von Code für zusätzliche Laufzeittests gesteuert werden kann. Beim Beginn der Übersetzung ist der Schalter standardmäßig auf *aus* gestellt. Nach einem aktiven Kommentar der Form

```
(*$R+   *)
```

steht der Schalter für Bereichstests (*range check*) auf *ein*. Der Compiler erzeugt dann Codes, um bei der Laufzeit des Programmes auch folgende Operationen zu prüfen:

Zusätzliche Bereichsüberprüfungen zur Laufzeit

- Zuweisungen von Werten eines Grundtyps an Variablen eines Unterbereichstyps. Dies betrifft auch die Grenzen von FOR..DO-Schleifen, bei denen die Laufvariable einen Unterbereichstyp besitzt. Da der Typ BYTE in Pascal 2.0 ebenfalls als Unterbereichstyp (0..255) deklariert ist, werden auch Zuweisungen an Variablen vom Typ BYTE geprüft.

  Beispiel:

  Bei Zuweisungen von Werten des Typs INTEGER an die Variable X wird geprüft, ob der Wert außerhalb des Bereiches 1900 bis 1986 liegt:

```
VAR X,Y: 1900..1986;
(*$R+ *)
X:= X + 30;          (* Ergebnis wird geprüft  *)
X:= 2000;            (* --> Laufzeitfehler     *)
READ(X);             (* Eingabe wird geprüft   *)
X:= Y;               (* keine Prüfung          *)
FOR X:= X+4 TO Y DO  (* nur der Startwert wird *)
                     (* geprüft                *)
(*$R- *)
X:= 2000;            (* keine Prüfung          *)
```

- Mengenoperationen der Form
  `[A] [A..B] und B IN X`
  werden auf gültige Werte von A und B (d.h. ORD(A) und ORD(B) aus dem Bereich 0 bis 95) geprüft.
  Beispiel:
  ```
  VAR S,T: SET OF CHAR;
      C  : CHAR;
  (*$R+ *)
  S:= ['A'..'Z'];          (* wird überprüft          *)
  S:= S + ['a'..'z'];      (* --> Laufzeitfehler, da *)
                           (* ORD('a')>96 ist         *)
  S:= [CH];                (* wird überprüft          *)
  IF CH IN S THEN ...      (* CH wird überprüft       *)
  (*$R- *)
  IF 'a' IN S THEN ...     (* wird nicht überprüft,   *)
                           (* das Ergebnis ist FALSE  *)
  ```
- Array-Indizierungen für Felder der Form ARRAY [A..B] OF ... werden überprüft. Bei der Indizierung einer String-Variablen mit der Maximallänge Maxlen wird geprüft, ob der Index den Bereich 0..Maxlen verläßt.
  Beispiel:
  ```
  TYPE INDEX: -30..30;
  VAR F: ARRAY[INDEX] OF CHAR;
      S: STRING;
      T: STRING[30];
      X: INTEGER;
      I: INDEX;
  (*$R+ *)
  F[3]:= 'X';              (* wird geprüft          *)
  WRITE(F[I]);             (* keine Prüfung         *)
  IF F[X]='1' THEN ...     (* X wird geprüft        *)
  F[50]:= F[1];            (* --> Laufzeitfehler    *)
  S[80]:= 'c';             (* wird geprüft (OK)     *)
  S[I]:= CHR(5);           (* wird geprüft          *)
  T[80]:= 'c';             (* --> Laufzeitfehler    *)
  ```
- Schließlich werden noch die Ergebnisse der Standardprozeduren SUCC(X) und PRED(X) geprüft, falls X ein Ausdruck eines Aufzählungs- oder Unterbereichstyps ist.
  Beispiel:
  ```
  VAR TAG: (MONTAG,DIENSTAG,MITTWOCH,DONNERSTAG,FREITAG,
            SAMSTAG,SONNTAG);
  ```

```
    WOTAG: MONTAG..FREITAG;
    X  : INTEGER;
(*$R+ *)
TAG:=SUCC(TAG));          (* wird geprüft           *)
WOTAG:= TAG;              (* wird geprüft           *)
WOTAG:= SUCC(WOTAG);      (* wird geprüft           *)
TAG:= SUCC(WOTAG);        (* wird geprüft           *)
X:= SUCC(X);              (* Zuweisung wird geprüft *)
```

Tritt in einem dieser Fälle ein ungültiger Wert auf, so wird die Programmausführung mit der Fehlermeldung

Fehlermeldung bei Bereichsüberschreitung

```
VALUE OUT OF BOUNDS: x          low   high
```

abgebrochen. Dabei stellen x, low und high die Ordinalwerte des fehlerhaften Wertes, der unteren und der oberen Bereichsgrenze dar.

Beispiel:

Nach der obigen Deklaration der Variablen WOTAG wird bei der ungültigen Zuweisung WOTAG:= SAMSTAG folgende Fehlermeldung ausgegeben:

```
VALUE OUT OF BOUNDS: 5  0  4
```

Es gilt nämlich:

```
ORD(SAMSTAG) = 5
ORD(MONTAG)  = 0
ORD(FREITAG) = 4
```

Der *Schalter* für Bereichsüberprüfung kann mit dem folgenden aktiven Kommentar wieder auf *aus* geschaltet werden:

```
(*$R- *)
```

Durch das Einschalten der Bereichsüberprüfungsoption wachsen die Codegröße und Programmlaufzeit nur minimal. Zumindest während der Programmentwicklung sollten Sie daher diese Option wählen, um verborgene illegale Zuweisungen festzustellen.

4.4.6.2 Parameterüberprüfung

Prüfung der String-Variablenparameter

Pascal 2.0 erlaubt eine Lockerung der Typüberprüfung für Variablenparameter des Typs STRING. Durch die Speicherungsform von Strings ist es nämlich sinnvoll, an eine Prozedur oder Funktion als aktuellen Parameter eine String-Variable größerer Maximallänge als der zugehörige formale Parameter zu übergeben. Dies widerspricht jedoch den im *report* vereinbarten Typkompatibilitätsregeln, nach denen aktuelle *Variablenparameter* denselben Typ wie der formale Parameter besitzen müssen.

Nach dem aktiven Kommentar

```
(*$P-    *)
```

wird die standardmäßig eingeschaltete Typüberprüfung für String-Variablenparameter so gelockert, daß aktuelle Parameter länger als die formalen

Parameter-Strings sein dürfen. Um wieder kompatibel zum *report* zu sein, kann die Parameterüberprüfung anschließend wieder mit

```
(*$P+    *)
```

eingeschaltet werden.

Beispiel:

```
PROGRAM STRINGPARAMETER(INPUT,OUTPUT);
  VAR LONGSTRING: STRING[132];

  PROCEDURE UPCASE(VAR S: STRING);
  (* wandele S in Großbuchstaben *)
       VAR L: INTEGER;
  BEGIN
    FOR L:= 1 TO LENGTH(S) DO
         S[L]:= CHR(ORD(S[L]) AND 127);
  END; (* UPCASE *)

BEGIN
  UPCASE(LONGSTRING);     (* Fehlermeldung *)
  (*$P- *)
  UPCASE(LONGSTRING);     (* keine Fehlermeldung *)
  (*$P+ *)
  UPCASE(LONGSTRING);     (* Fehlermeldung *)
END.
```

4.4.6.3 Chain-Option

Unterdrücken der Namen in der dynamic chain

Standardmäßig erzeugt der Compiler *Pascal 2.0* für jeden Block Codes, um im Falle einer Fehlermeldung zur Programmlaufzeit die dynamische Aufrufkette (*dynamic chain*) anzuzeigen. In Abschnitt 4.3.3 wurde folgendes Beispiel einer Laufzeitfehlermeldung gegeben.

```
DIVISION BY ZERO
 ERROR AT 7437 IN KEHRWERT
CALLED AT 7453 IN P
CALLED AT 7469 IN LAUFZEITFEHLER
```

Für jeden Block (Hauptprogramm, Prozedur, Funktion) wird ihr Name einmal gespeichert. Möchten Sie diesen Speicherplatz einsparen oder einem Benutzer Ihrer übersetzten Programme keinen Einblick in die Programmstruktur geben, so können Sie den Schalter zur Speicherung der Namen auf *aus* stellen:

```
(*$C-    *)
```

Dieser aktive Kommentar muß vor dem Programm-, Prozedur- oder Funktionskopf stehen, um für einen Block gültig zu sein. Sie können den Schalter

auch selektiv für einzelne Blöcke ein- und ausschalten, dann treten in der *dynamic-chain* teilweise Fragezeichen »?« als Blocknamen auf:

Beispiel:

```
(*$C- *)
PROGRAM LAUFZEITFEHLER(INPUT,OUTPUT);

FUNCTION KEHRWERT(X: INTEGER): REAL;
BEGIN
  KEHRWERT:= 1 / X;
END; (* KEHRWERT *)

(*$C+ *)
PROCEDURE P;
BEGIN
  WRITELN(KEHRWERT(0));
END; (* P *)

BEGIN
  P
END.
```

Die Division durch null führt zur folgenden Fehlermeldung:

```
DIVISION BY ZERO
 ERROR AT 7437 IN ?
CALLED AT 7453 IN P
CALLED AT 7469 IN ?
```

Die Chain-Option hat keinerlei Auswirkung auf die Ausführungsgeschwindigkeit übersetzter Pascal-Programme.

4.4.6.4 Include-Files

Aufteilung langer Quelltexte in Include-Files

Wie bereits in den vorangegangenen Abschnitten erwähnt, kann der Compiler Teile des Quelltextes bei der Übersetzung von Diskette lesen. Dies ist immer dann notwendig, wenn die Größe des Quelltextes eine Größe von 22 Kbyte überschreitet. Aber auch um logisch zusammengehörige Teile zusammenzufassen oder um den Quelltext überschaubar zu halten, bietet sich die Auslagerung von Texten in Include-Files an.

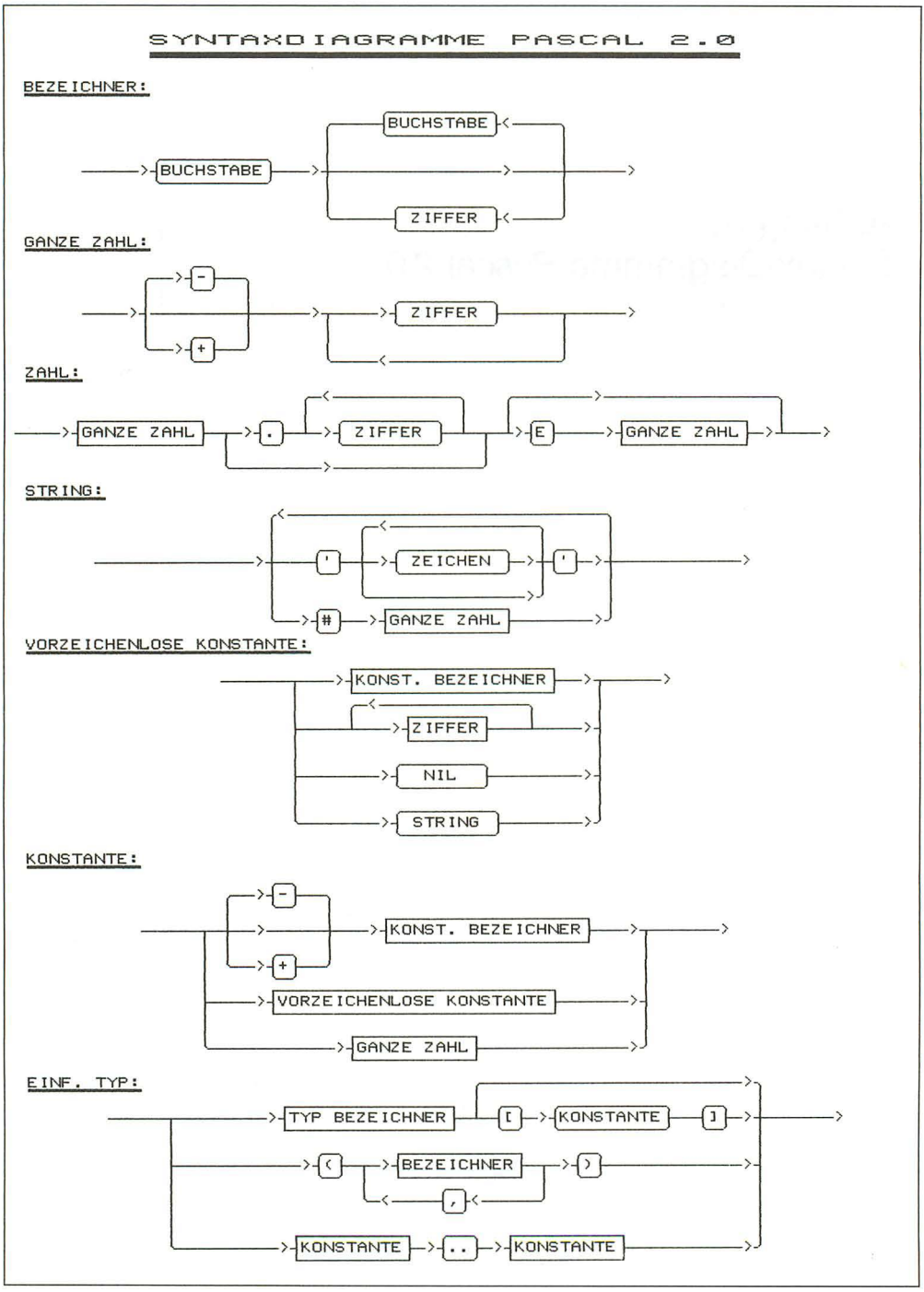
Um Include-Files effizient zur Modularisierung von Programmen einsetzen zu können, ist in *Pascal 2.0* die Reihenfolge der Deklarationsteile beliebig. Außerdem kann jeder Deklarationsteil beliebig oft auftreten. Somit können Sie im Vereinbarungsteil des Hauptprogrammes Include-Files aufrufen, die jeweils Konstanten, Typen, Variablen und Prozeduren *exportieren*.

```
PROGRAM INCLUDES(INPUT,OUTPUT);
(*$"TURTLE.INC" *)
(*$"STRINGS.INC" *)
BEGIN
END.
```

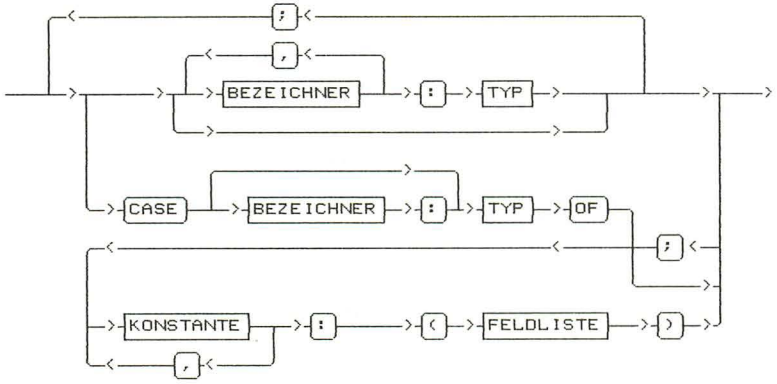
Erreicht der Compiler bei der Übersetzung einen aktiven Kommentar, der direkt nach dem Dollarzeichen einen File-Namen in Anführungszeichen enthält, so wird das Ende dieser Zeile ignoriert und ab dieser Stelle der Programmtext von dem File (z.B. »TURTLE.INC«) auf der eingelegten Diskette gelesen. Am Dateiende setzt der Compiler die Übersetzung mit der Folgezeile im Arbeitsspeicher fort.

Include-Files sind normale Quelltexte, die mit den *primary-commands* SAVE oder END des Editors auf Diskette gespeichert wurden. In einem Programm können beliebig viele Include-Files aufgerufen werden. Außerdem kann auch in einem Include-File ein anderes Include-File aufgerufen werden. Jedoch sind *keine* Schachtelungen möglich, da der Compiler am Ende eines Files immer zur Übersetzung aus dem Speicher zurückkehrt.

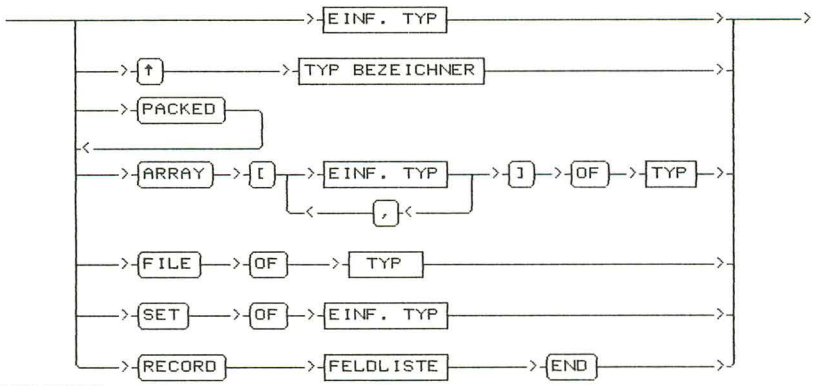
Anhang A: Syntax-Diagramme Pascal 2.0


SYNTAXDIAGRAMME PASCAL 2.0
BEZEICHNER:
BUCHSTABE
BUCHSTABE
ZIFFER
GANZE ZAHL:
-
+
ZIFFER
ZAHL:
GANZE ZAHL
.
ZIFFER
E
GANZE ZAHL
STRING:
'
ZEICHEN
'
#
GANZE ZAHL
VORZEICHENLOSE KONSTANTE:
KONST. BEZEICHNER
ZIFFER
NIL
STRING
KONSTANTE:
-
+
KONST. BEZEICHNER
VORZEICHENLOSE KONSTANTE
GANZE ZAHL
EINF. TYP:
TYP BEZEICHNER
[
KONSTANTE
]
(
BEZEICHNER
,
)
KONSTANTE
..
KONSTANTE

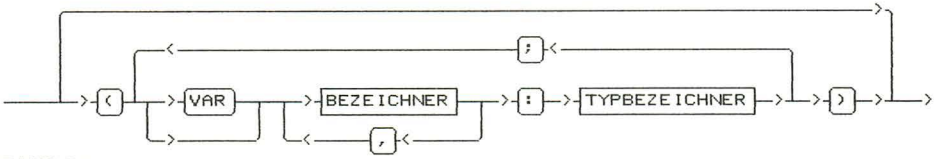
FELDLISTE:



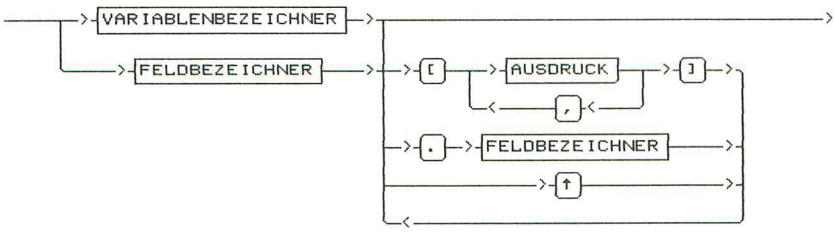
TYP:

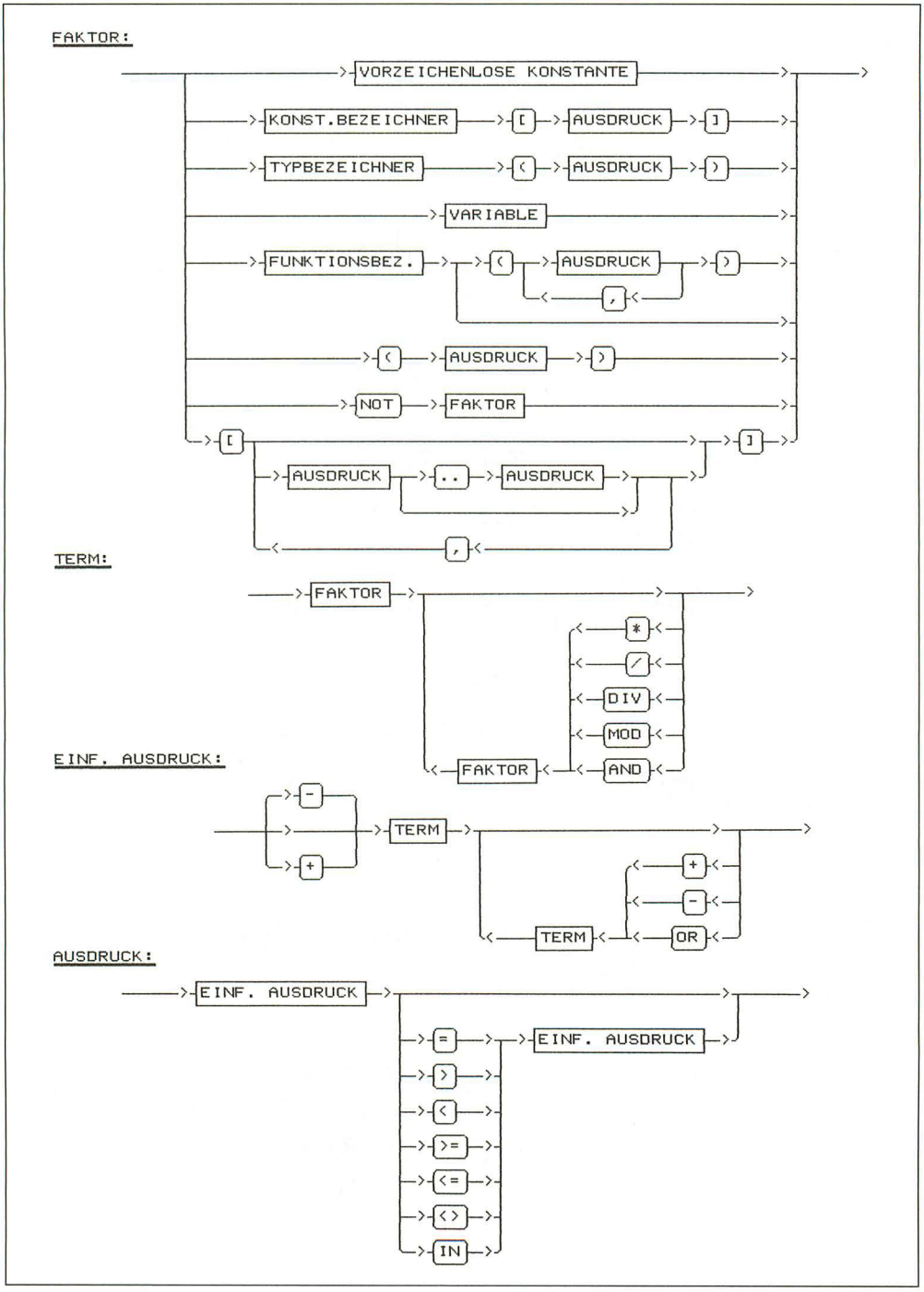


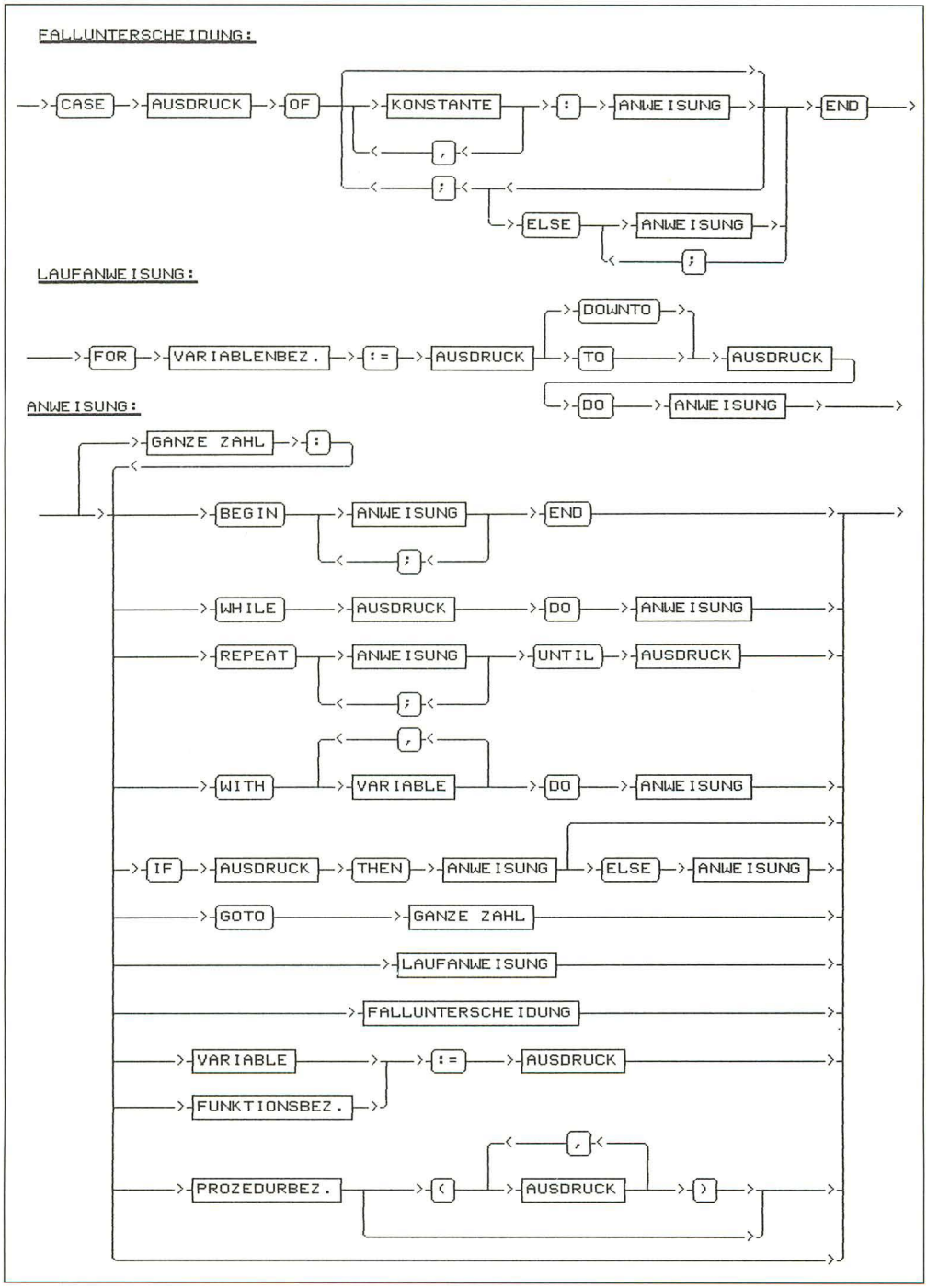
PARAMETERLISTE:

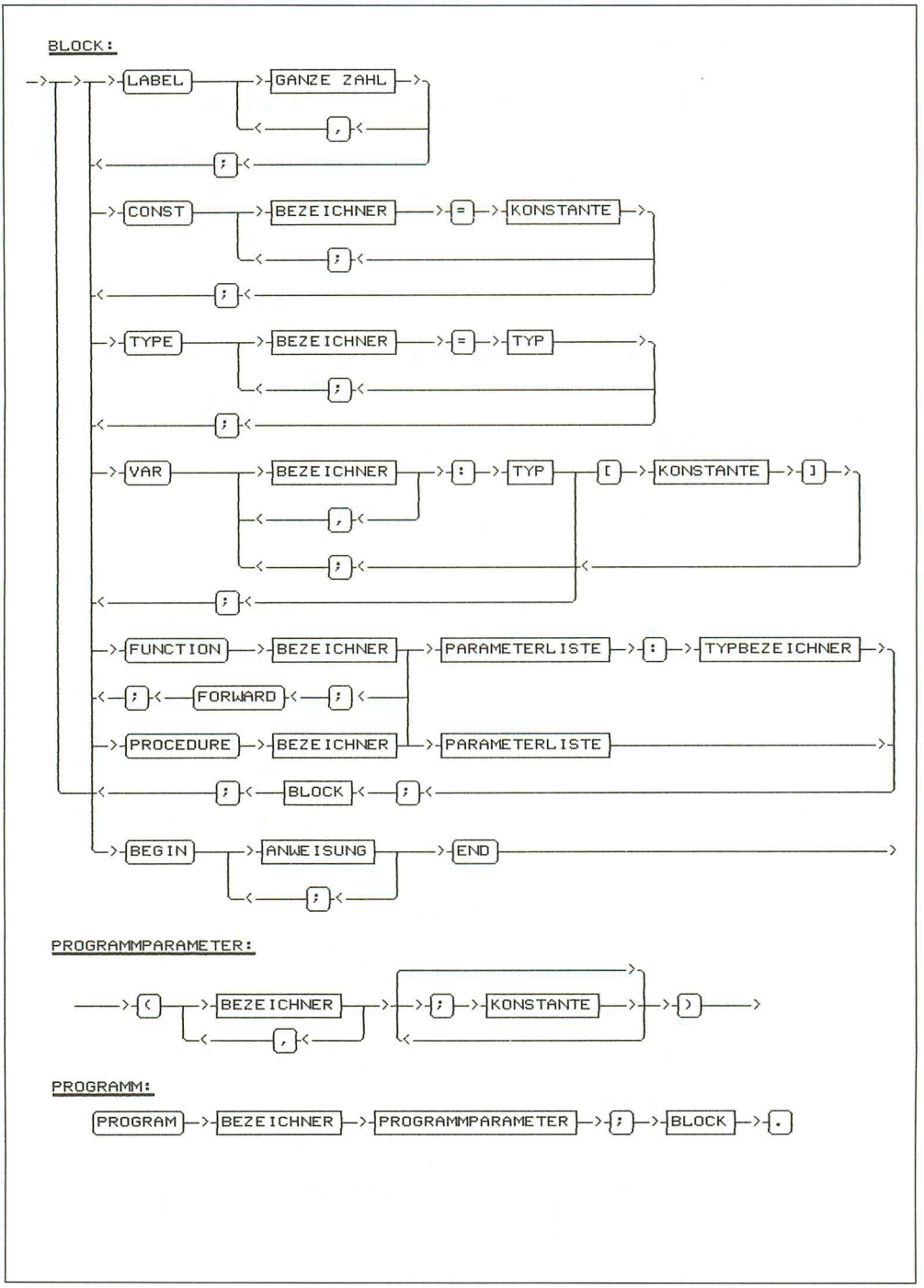


VARIABLE:




FAKTOR:
VORZEICHENLOSE KONSTANTE
KONST.BEZEICHNER
[
AUSDRUCK
]
TYPBEZEICHNER
(
AUSDRUCK
)
VARIABLE
FUNKTIONSBEZ.
(
AUSDRUCK
,
)
(
AUSDRUCK
)
NOT
FAKTOR
[
AUSDRUCK
..
AUSDRUCK
,
]
TERM:
FAKTOR
*
/
DIV
MOD
AND
FAKTOR
EINF. AUSDRUCK:
-
+
TERM
+
-
OR
TERM
AUSDRUCK:
EINF. AUSDRUCK
=
>
<
>=
<=
<>
IN
EINF. AUSDRUCK


FALLUNTERSCHEIDUNG:
CASE
AUSDRUCK
OF
KONSTANTE
,
:
ANWEISUNG
;
ELSE
ANWEISUNG
;
END
LAUFANWEISUNG:
FOR
VARIABLENBEZ.
:=
AUSDRUCK
DOWNTO
TO
AUSDRUCK
DO
ANWEISUNG
ANWEISUNG:
GANZE ZAHL
:
BEGIN
ANWEISUNG
;
END
WHILE
AUSDRUCK
DO
ANWEISUNG
REPEAT
ANWEISUNG
;
UNTIL
AUSDRUCK
WITH
,
VARIABLE
DO
ANWEISUNG
IF
AUSDRUCK
THEN
ANWEISUNG
ELSE
ANWEISUNG
GOTO
GANZE ZAHL
LAUFANWEISUNG
FALLUNTERSCHEIDUNG
VARIABLE
:=
AUSDRUCK
FUNKTIONSBEZ.
PROZEDURBEZ.
(
,
AUSDRUCK
)


BLOCK:
LABEL
GANZE ZAHL
,
;
CONST
BEZEICHNER
=
KONSTANTE
;
;
TYPE
BEZEICHNER
=
TYP
;
;
VAR
BEZEICHNER
:
TYP
[
KONSTANTE
]
,
;
;
FUNCTION
BEZEICHNER
PARAMETERLISTE
:
TYPBEZEICHNER
;
FORWARD
;
PROCEDURE
BEZEICHNER
PARAMETERLISTE
;
BLOCK
;
BEGIN
ANWEISUNG
;
END
PROGRAMMPARAMETER:
(
BEZEICHNER
,
;
KONSTANTE
)
PROGRAMM:
PROGRAM
BEZEICHNER
PROGRAMMPARAMETER
;
BLOCK
.

Anhang B: Fehlernummern Pascal 2.0

0 Position eines Laufzeitfehlers (Option LOCATE ADDRESS)
1 Fehler in Typangabe
2 Bezeichner erwartet
3 'PROGRAM' erwartet
4 ')' erwartet
6 ':' erwartet
8 'OF' erwartet
9 '(' erwartet
11 '[' erwartet
12 ']' erwartet
13 'END' erwartet
14 ';' erwartet
15 INTEGER erwartet
16 '=' erwartet
17 'BEGIN' erwartet
20 '.' erwartet
21 ',' erwartet
22 Fehler in String-Konstante (z.B. String zu lang)
50 Fehler in Konstante
51 ':=' erwartet
52 'THEN' erwartet
53 'UNTIL' erwartet
54 'DO' erwartet
55 'TO' oder 'DOWNTO' erwartet
56 Fehler im Vereinbarungsteil
59 Fehler in Variable (Variablenbezeichner erwartet)
60 String ist hier nicht zulässig
101 Bezeichner zweimal deklariert
102 Untere Grenze übersteigt obere Grenze
103 Bezeichner ist nicht von der richtigen Klasse

104 Bezeichner ist nicht deklariert
105 Vorzeichen hier nicht erlaubt
106 Zahl erwartet
107 Inkompatible Unterbereichstypen
108 Ziel-String zu kurz für Zuweisung dieses Wertes
109 Grundtyp muß Skalartyp oder Unterbereich sein (nicht REAL)
110 Typ des *tagfield* muß Skalartyp oder Unterbereich sein
111 Konstante ist nicht kompatibel mit dem *tagfield*
112 String-Ausdruck erwartet
113 Indextyp muß Skalartyp oder Unterbereich sein (weder REAL noch INTEGER)
114 Maximale String-Länge größer als 255 Zeichen
116 Falscher Typ eines Parameters für Standardprozeduren
117 Ungelöste Vorwärtsvereinbarung eines Zeigertypbezeichners
118 Undeklarierter Typbezeichner in dieser Variablendeklaration
119 Nicht alle FORWARD-Deklarationen gelöst
120 Typbezeichner für Ergebnistyp der Funktion erwartet
121 File als Wertparameter nicht zulässig
123 Ergebnistyp fehlt im Funktionskopf
125 Falscher Typ eines Parameters für Standardfunktionen
126 Anzahl der Parameter stimmt nicht mit der Deklaration überein
127 Unzulässige Parameter-Substitution
129 Operandentypen nicht kompatibel
130 Ausdruck ist nicht vom Typ Menge (SET OF ...)
131 Nur Test auf Gleichheit und Ungleichheit zulässig
132 Test auf echtes Enthaltensein nicht zulässig
133 Variable ist nicht vom Typ STRING
134 Unzulässiger Operandentyp
136 Elementtyp einer Menge muß Skalartyp oder Unterbereich sein
137 Elementtypen nicht kompatibel
138 Variable ist nicht vom Typ ARRAY oder STRING
139 Indextyp entspricht nicht der Deklaration
140 Variable ist nicht vom Typ RECORD
141 Variable ist weder Zeiger noch FILE
142 Ausdruck ist nicht vom Typ INTEGER
143 Laufvariable besitzt einen unzulässigen Typ
144 Ausdruck hat einen unzulässigen Typ
145 Typkonflikt
146 Zuweisungen zwischen Files sind nicht möglich
147 Typ der Fallmarke nicht kompatibel mit CASE-Ausdruck
148 Ausdruck vom Typ BOOLEAN erwartet

152 Ein solches Feld existiert nicht in diesem Record
154 Aktueller Parameter muß eine Variable sein
155 Laufvariable muß eine lokale Variable sein
159 REAL oder STRING nicht als *tagfield* zulässig
161 FORWARD ist hier nicht zulässig
165 Label mehrfach (im Anweisungsteil) definiert
166 Label mehrfach (im Vereinbarungsteil) deklariert
167 Label nicht deklariert
168 Undefiniertes Label im vorherigen Block
171 Variable muß vom Typ FILE sein
172 Fehlender Parameter für Standardprozedur
173 File ist nicht vom Typ TEXT (PUT oder GET benutzen!)
174 Standard-File wiederdefiniert
175 Fehler in Anweisungsfolge
400 Zu viele Fallmarken in CASE-Anweisung
401 Zu viele Labels im Programm
402 Zu viele Bezeichner im Programm
500 Operandentypen müssen INTEGER sein
501 Ordnungszahlen des Grundtyps liegen nicht im Bereich 0 bis 95
502 Typ BOOLEAN oder INTEGER erwartet
503 Externe Files werden hier nicht angegeben
505 Standard-File OUTPUT muß deklariert werden
506 'NIL' ist hier nicht zulässig
507 Registervariable bei SYS muß länger als 4 Byte sein
510 FORWARD-Deklaration muß in der gleichen statischen Schachtelungstiefe erfolgen
511 Ganze Zahl erwartet
512 Parameter dürfen keine absolute Adresse erhalten

Weitere Meldungen im Compiler:

NO MESSAGE FOUND
Diese Meldung wird ausgegeben, falls die Datei ERRORS.TXT auf der eingelegten Diskette nicht gefunden wurde oder in dieser Datei zur vorliegenden Fehlernummer kein Text existiert.

BREAK
Diese Meldung wird ausgegeben, falls während der Kompilation die Run/Stop-Taste betätigt wurde.

PROGRAM INCOMPLETE
Diese Meldung wird ausgegeben, falls unerwartet das Ende des Quelltextes erreicht wurde. Die Übersetzung wird abgebrochen.

EDITOR OVERWRITTEN
SOURCE OVERWRITTEN
SYSTEM SWITCHED OFF
Diese Meldungen werden ausgegeben, wenn durch das erzeugte Objektprogramm Teile des Quelltextes oder Editors überschrieben wurden. Nähere Einzelheiten finden sich in Abschnitt 4.3.2.

Fehlertexte im File ERRORS.TXT

```
  0 ADDRESS OF RUNTIME ERROR FOUND
  1 ERROR IN SIMPLE TYPE
  2 IDENTIFIER EXPECTED
  3 'PROGRAM' EXPECTED
  4 ')' EXPECTED
  5 ':' EXPECTED
  8 'OF' EXPECTED
  9 '(' EXPECTED
 11 '[' EXPECTED
 12 ']' EXPECTED
 13 'END' EXPECTED
 14 ';' EXPECTED
 15 INTEGER EXPECTED
 16 '=' EXPECTED
 17 'BEGIN' EXPECTED
 20 '.' EXPECTED
 22 ',' EXPECTED
 23 ERROR IN STRING CONSTANT (E.G. STRING TOO LONG)
 50 ERROR IN CONSTANT
 51 ':=' EXPECTED
 52 'THEN' EXPECTED
 53 'UNTIL' EXPECTED
 54 'DO' EXPECTED
 55 'TO/DOWNTO' EXPECTED
 56 ERROR IN DECLARATION PART
 59 ERROR IN VARIABLE (VARIABLE IDENTIFIER EXPECTED)
 60 STRING NOT ALLOWED HERE
101 IDENTIFIER DECLARED TWICE
102 LOW BOUND EXCEEDS HIGH BOUND
103 IDENTIFIER IS NOT OF APPROPRIATE CLASS
104 IDENTIFIER NOT DECLARED
105 SIGN NOT ALLOWED HERE
106 NUMBER EXPECTED
107 INCOMPATIBLE SUBRANGE TYPES
108 TARGET STRING IS TOO SHORT
109 BASE TYPE HAS TO BO SCALAR OR SUBRANGE (NOT REAL)
110 TAGFIELD TYPE HAS TO BE SCALAR OR SUBRANGE
111 CONSTANT INCOMPATIBLE WITH TAGFIELD TYPE
112 STRING EXPRESSION EXPECTED
113 INDEX TYPE HAS TO BE SCALAR OR SUBRANGE (NEITHER REAL NOR INTEGER)
114 MAX. STRING SIZE IS 255 CHARS
116 ERROR IN TYPE OF STANDARD PROCEDURE PARAMETER
117 INCOMPLETE POINTER TYPE IN PRECEDING DECLARATIONS
118 FORWARD REFERENCE TYPE IDENTIFIER IN VARIABLE DECLARATION
119 UNSATISFIED 'FORWARD' DECLARATION IN PROGRAM
120 TYPE IDENTIFIER (FUNCTION RESULT) EXPECTED
121 FILE VALUE PARAMETERS NOT ALLOWED
123 MISSING RESULT TYPE IN FUNCTION DECLARATION
125 ERROR IN TYPE OF STANDARD FUNCTION PARAMETER
126 NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION
```

```
127 ILLEGAL PARAMETER SUBSTITUTION
129 TYPE CONFLICT OF OPERANDS
130 EXPRESSION IS NOT OF SET TYPE
131 TEST ON EQUALITY ALLOWED ONLY
132 STRICT INCLUSION NOT ALLOWED
133 VARIABLE IS NOT OF TYPE STRING
134 ILLEGAL TYPE OF OPERANDS
136 SET ELEMENT TYPE HAS TO BE SCALAR OR SUBRANGE
137 SET ELEMENT TYPES NOT COMPATIBLE
138 TYPE OF VARIABLE IS NOT ARRAY
139 INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION
140 TYPE OF VARIABLE IS NOT RECORD
141 TYPE OF VARIABLE IS NOT POINTER OR FILE
142 EXPRESSION IS NOT OF TYPE INTEGER
143 ILLEGAL TYPE OF LOOP CONTROL VARIABLE
144 ILLEGAL TYPE  OF EXPRESSION
145 TYPE CONFLICT
146 ASSIGNATION OF FILES NOT ALLOWED
147 LABEL TYPE INCOMPATIBLE WITH SELECTING EXPRESSION
148 BOOLEAN EXPRESSION EXPECTED
152 NO SUCH FIELD IN THIS RECORD
154 ACTUAL PARAMETER HAS TO BE A VARIABLE
155 CONTROL VARIABLE HAS TO BE A LOCAL UNPACKED VARIABLE
159 REAL OR STRING TAGFIELDS NOT ALLOWED
161 AGAIN FORWARD DECLARED ('FORWARD' NOT ALLOWED HERE)
165 MULTIDEFINED LABEL
166 MULTIDECLARED LABEL
167 UNDECLARED LABEL
168 UNDEFINED LABEL (IN PREVIOUS BLOCK)
171 VARIABLE HAS TO BE OF TYPE FILE
172 MISSING STANDARD PROCEDURE PARAMETER
173 FILE IS NOT OF TYPE TEXT (USE PUT OR GET!)
174 STANDARD FILE REDECLARED
175 ERROR IN STATEMENT SEQUENCE
400 TOO MANY CASE LABELS IN CASE STATEMENT
401 TOO MANY LABELS IN PROGRAM
402 TOO MANY IDENTIFIERS IN PROGRAM
500 TYPE OF OPERAND(S) HAS TO BE INTEGER
501 ORDINAL VALUE OF BASE TYPE OUT OF RANGE [0..95]
502 TYPE BOOLEAN OR INTEGER EXPECTED
503 NO DECLARATION OF EXTERNAL FILES ALLOWED HERE
504 VARIABLE PARAMETER MUST NOT BE PACKED
505 AT LEAST 1 STANDARD FILE HAS TO BE DECLARED
506 'NIL' NOT ALLOWED HERE
507 VARIABLE IS TOO SMALL (<4 BYTES) FOR REGISTERS
510 FORWARD DECLARATION HAS TO BE SATISFIED ON THE SAME LEVEL
511 INTEGER NUMBER EXPECTED
512 ABSOLUTE ADDRESSED PARAMETERS NOT ALLOWED
```

Anhang C: Laufzeitfehler

Bei der Ausführung eines Programmes auftretende Fehler werden im Klartext gemeldet. Die Bedeutung der angezeigten *dynamic chain* und Methoden zur Lokalisierung des Fehlers werden in Abschnitt 4.3.3 erklärt.

STACK OVERFLOW	Am Prozeduranfang: Es existiert kein Speicherplatz für lokale Variablen. Sonst: Kein Speicherplatz für Zwischenergebnisse
INTEGER OVERFLOW	Bereichsüberschreitung bei Operationen mit ganzen Zahlen oder grober Indizierungsfehler
DIVISION BY 0	Division durch Null bei den Operationen MOD und DIV
NO LABEL IN CASE	Keine Fallmarke oder ELSE-Zweig in CASE-Anweisung gefunden
HEAP OVERFLOW	Bei der Standardprozedur NEW ist auf dem *heap* kein Platz für eine neue dynamische Variable vorhanden
STRING OVERFLOW	Bei String-Operationen wird ein Zwischenergebnis länger als 255 Zeichen, bei Zuweisungen wird die Länge des Ziel-Strings überschritten, oder bei Prozeduraufrufen ist der aktuelle Parameter zu lang

VALUE OUT OF BOUNDS	In einem Ausdruck tritt ein illegaler Wert auf (siehe Abschnitt 4.4.6.1). Es werden außerdem folgende Ordinalwerte ausgegeben:
	`ORD(fehlerhafter Wert)`
	`ORD(untere Bereichsgrenze)`
	`ORD(obere Bereichsgrenze)`

BREAK	Das Programm wurde mit der Run/Stop- und Restore-Taste unterbrochen

TOO MANY FILES OPEN	Es dürfen maximal 10 Files gleichzeitig geöffnet sein (CLOSE nicht vergessen!)
FILE NOT FOUND	Bei OPEN konnte das angegebene File nicht gefunden werden
DEVICE NOT PRESENT	Bei READ, WRITE, GET oder PUT wurde festgestellt, daß das Peripheriegerät nicht aktiv ist
NOT INPUT FILE	Dieses Gerät (z.B. der Bildschirm) kann keine Daten liefern
NOT OUTPUT FILE	An dieses Gerät (z.B. die Tastatur) kann man keine Daten senden
MISSING FILE NAME	Bei OPEN muß bei diesem Gerät ein File-Name angegeben werden
ILLEGAL DEVICE NUMBER	Diese Geräteadresse beim OPEN-Befehl ist nicht zulässig
ILLEGAL QUANTITY	Beim Aufruf einer Standardfunktion oder -prozedur wurden illegale reelle oder ganzzahlige Argumente übergeben. Bei der Option LOCATE ADDRESS wird das erste Symbol **nach** der Parameterliste markiert!
OVERFLOW	Bei einer Operation mit reellen Zahlen trat eine Bereichsüberschreitung auf
DIVISION BY ZERO	Bei einer Division (mit /) ist der zweite Operand 0.0
NO GRAPHICS AREA	Bei einem Befehl für den Grafikbildschirm (BOX, PAINT, CIRCLE, DISPLAY) wurde festgestellt, daß kein Speicherplatz für eine *bit-map* reserviert wurde. Dies geschieht, falls im Programmkopf der Name GRAPHIC fehlt.

Anhang D: Operatoren in Pascal 2.0

Operator	Operation	Operandentypen	Ergebnistyp
+ (Vorzeichen)	Identität	Integer, Real	wie Operand
– (Vorzeichen)	Vorzeichenumkehr	Integer, Real	wie Operand
+	Addition	Integer, Real	Integer, Real
	Vereinigungsmenge	Menge	Menge
	Konkatenation	String	String
–	Subtraktion	Integer, Real	Integer, Real
	Differenzmenge	Menge	Menge
*	Multiplikation	Integer, Real	Integer, Real
	Schnittmenge	Menge	Menge
DIV	Division mit Rest	Integer	Integer
MOD	Divisionsrest	Integer	Integer
/	Division	Integer, Real	Real
=	gleich	Skalar, Pointer, Menge, String	Boolean
<>	ungleich	Skalar, Pointer, Menge, String	Boolean
<	kleiner	Skalar, String	Boolean
>	größer	Skalar, String	Boolean
<=	kleiner oder gleich	Skalar, String	Boolean
	Test auf Teilmenge	Menge	Boolean
>=	größer oder gleich	Skalar, String	Boolean
	Test auf Obermenge	Menge	Boolean
IN	Test auf Enthaltensein	1. Operand Skalar 2. Operand Menge	Boolean
NOT	nicht	Boolean	Boolean
	1er-Komplement	Integer	Integer
OR	oder	Boolean	Boolean
	bitweise ODER	Integer	Integer
AND	und	Boolean	Boolean
	bitweise UND	Integer	Integer

Bemerkung:

Die Verwendung von INTEGER-Operanden bei den logischen Operatoren NOT, AND und OR ist nicht im *report* erlaubt. String-Konkatenation mit dem Plus-Symbol ist dort ebenfalls nicht definiert.

Anhang E: Literaturhinweise

1. Jensen, K., Wirth, N.: PASCAL, User Manual and Report
   Lecture Notes in Computer Science, Vol. 18.
   Springer-Verlag 1974
2. Wirth, N.: Algorithmen und Datenstrukturen
   LAMM Teubner-Studienbücher Informatik 1979
3. Barron, D. W. (Editor): PASCAL – The Language and its Implementation
   Wiley Series in Computing, John Wiley and Sons 1981
4. Harrington, S.: Computer Graphics (A Programming Approach)
   International Student Edition, Mc Graw-Hill 1983
5. Wirth, N.: Systematisches Programmieren. Eine Einführung
   LAMM Teubner-Studienbücher Informatik 1972
6. Knuth, D.E.: The Art of Computer Programming. Band 1 und 3
   Addison Wesley 1973
7. Maurer, H.: Datenstrukturen und Programmierverfahren
   LAMM Teubner-Studienbücher Informatik 1974
8. Mehlhorn, K.: Effiziente Algorithmen
   LAMM Teubner-Studienbücher Informatik 1977
9. Wirth, N.: Compilerbau. Eine Einführung
   LAMM Teubner-Studienbücher 1981
10. Gerits, Schieb, Thrun: Commodore 128 Intern
    DATA BECKER GmbH
11. Commodore 128/128D Bedienungshandbuch
    Commodore Büromaschinen GmbH Frankfurt 1985
12. Matthes, F.: Pascal mit dem C64
    Markt&Technik Verlag 1986

13. Matthes, F.: Pascal-Kurs
Sonderheft Programmiersprachen 5/1986 Happy-Computer
Markt&Technik Verlag

14. Wirth, N.: Programmieren in Modula-2
Springer Verlag 1985

15. Kröpelin, R.: CBM-DOS mit Tücken
Zeitschrift mc 8/1986

Stichwortverzeichnis

80-Zeilen-Modus 273

A
Absolute Adresse 225
Absolute Variable 226, 293
Algorithmus 152
AND 292
Anweisung 56
–, bedingte 58
Anweisungsfolge 56
APPEND 230
Arbeitsdiskette 14, 16
Array 80, 83
–, eindimensional 80
–, Indizierung 81
–, mehrdimensional 93
ARRAY OF CHAR 89f.
Array-Operation 84
ASCII-Tabelle 177
Ausdruck 34, 48
Auswahl
–, einseitig 58
–, zweiseitig 58

B
BACKUP 230
Balkendiagramm 131
bank-switching 312
BASIC 11, 13
Baum 203
BEGIN 31, 33, 56, 59
Berechnung von Nullstellen 39
Bildschirm
–, Ausgabe 35
–, Eingabe 35
–, Funktion 70
Bildschirmspeicher 228
Binäre Suche 100
Blättern 18
BLOCK 31
Blockbefehl 99f., 255
Blockstruktur 55f., 63, 76
Blockwiederholung 64
Bogenmaß 45
BOOLEAN 50, 98
Boolesche Variable 63
Boolescher Ausdruck 51, 65
BOTTOM 248
Bottom-up 163
Bubblesort 103

C
C 128 – Modus 211
CANCEL 252
CASE 61f., 159, 292
CHAIN 332
CHANGE 254, 262
CHAR 39, 48f.
CLOSE 301
Codeerzeugung 273
COLLECT 230
COLUMNS 252
Compiler 11ff., 20, 91, 272
–, Codegröße 276
–, Fehlermeldung 275
–, Laufzeitfehler 275
–, Syntaxfehler 274
–, Zeilenfehler 275
CONCAT 231, 308
COPY 231, 255, 268, 309
Cursorbewegung 270
Cursorsteuerung 249

D
Datenstruktur 100, 185
–, linear 191
–, nichtlinear 203
–, variant 157
Datentyp 42
DEC 231
DELETE 308
Dereferenzierung 187
Directory 215
DIRECTORY 232, 254
DIV 44
DRIVE 254

E
Editor 12, 17, 245, 271
– verlassen 20
Editor-Kommando 19
Effizienzsteigerung 87
ELSE 59f., 292
END 31, 33, 56, 59, 252
EOF 165, 301
EOLN 179, 302

F
Fallunterscheidung 61
FALSE 63
Farbcode 318
Farbkonfiguration 16
Fehlerkorrektur 14
Fehlermeldung 22
Fehlernummern 341ff.
Feldgröße 37
Festkommadarstellung 45
File 163
File-Operation 166
Filetyp 213
File-Variable 212
Filter 322
FIND 254, 259
Fließkommazahlen 46
FOR 72
FOR ... NEXT 55
FOR-Schleife 73, 75
Formale Sprache 25
Formaler Parameter 113
Formatierung 37
– reeller Zahlen 38
FORWARD 129, 291
Freispeicherliste 201
Funktionen 116, 177
Funktionstasten 269

G
Geschwindigkeitsvergleich 47
GET 303
GOSUB 55
GOTO 55, 76
Grafik 131
Grafikmodus 314
Grafikprogramm 280
Grafikprozedur 314

H
HEADER 232
heap 226, 280, 298
HEX$ 232

I
IF 55ff.
INCLUDE-File 236
Indexprüfung 87
Index-Regeln 82
Indextyp 83
Indizierung 94

Initialisierung 205
INPUT 30, 35, 178, 255, 264
INSERT 308
Installationsmenü 15
INTEGER 39, 43, 47, 49, 292, 311
Integralberechnung 73
Interpreter 13
Iteration 126

K
Karteikasten 149
Kellerspeicher 192
KEY 233
Knotentyp 204
Kommandokanal 221
Kommentar 29
Konsistenz 55
Konstante 54
Kontrollstruktur 55
Kontrollzeichen 55
Koordinaten-Transformation 72
Kopieren 15, 182, 257
Kundenliste 198, 214

L
Laden 15
Laufvariable 73
Laufzeitfehler 13, 22, 273, 347ff.
Laufzeitsystem 12, 21, 225, 281
Leeranweisung 57
LENGTH 92, 309
line-command 255
Listentyp 191
LOCATE 253
Logische Datei 212
Logischer Operator 51
Lokale Deklaration 110
Lokaler Parameter 113
Lokalität 109
LOWER 253

M
Maschinenprogramm 282
Maschinensprache 224
Matrize 94f.
MAXINT 44
Mengenoperation 143
Mengentyp 142
Menüstruktur 242
MOD 44, 63
Modifikation
– der Speicherverteilung 282
– des Stack 284
MSKS 252

N
Netzgrafik 151
NEW 187
NIL 189
NOT 292
Nullstellen, Berechnung 39

O
Objektprogramm 13, 21, 278
– abbrechen 279
– speichern 277
– starten 277
ODD 50
OK 63
ON ... GOSUB 55
ON ... GOTO 55
OPEN 178, 212, 300
Operatoren 43, 51
OR 292
OUTPUT 30, 178, 255, 266

P
Parameter 35
– für Prozeduren 112
Parameterliste 130
Parameterüberprüfung 331
Pascal 11
–, Implementationen 284
–, Standardprozeduren 284
Pascal-Code 224
Pascal-Compiler 12, 20, 272
Pascal-Menü 243
Pascal-Routinen 229
Pascal-System 241, 279
PEEK und POKE 228
Permutation 125
Pixelcursor 314, 316
PLOT 71
Portabilität 12
PRED 136
primary-commands 251
Primzahl 146
PRINT 35
PROFILE 252
PROGRAMM 31
Programmkopf 30
Programmlisting 274
Programmsteuerung 47
Programmstruktur 29, 98
Programmteile nachladen 235
Prozedur 107f.
–, Aufruf 33
–, Parameter 112
Prozessorregister 227
Puffervariable 164
PUT 303

Q
Quelltext 13, 18, 22, 243, 333
QUICKSORT 127

R
READ 38ff., 91, 218, 303
READ(LN) 39f., 57, 95, 176
READIN 303
REAL 39, 44, 47
Record 148, 157, 160, 288
–, Eingabevariante 160
–, Speicherplatz 161
–, Speicherverteilung 160
RECORD 234
Recordnummer 218
Rekursion 121, 129, 205
REKURSIV 122
Relativdatei 218, 221f.
RENAME 234
REPEAT 63, 69f., 253
Report 284
RESET 252
Ringspeicher 196
Run/Stop-Taste 305

S
SAVE 252
Schachtelung 59, 111
Schlange 194
Schleifen 69
Scroll-Betrag 247
Sekundäradresse 212
Semikolon 57
Sequentielle Datei 163, 168
– lesen 165
– schreiben 164
– sortieren 169
–, Text ausgeben 266
–, Text einfügen 264
–, Files 255

Sicherungskopie 16, 23
Signifikante Stellen 46
Skalarer Typ 135, 142, 306
Sonderfälle 66
Sondertasten 249
Sonderzeichen 28, 295
Sortieren 84, 127
SOUND 321
Spaltenbereich 261
Spaltenmarkierung 248
Speicherorganisation in BASIC 276
Speicherverwaltung 121
Sprunganweisung 75, 290
stack 280
Stackanfang 239
Standardlösungen 130
Standardprozedur 120
Standardsprachumfang 25
Starten 16
STATUS 301
Statuszeile 247
Steuerzeichen 50, 177
STRING 52, 89
String-Konstante 27, 55
String-Operation 89, 91, 293, 308
Strings 27
–, Suche in 92
–, Vergleich zwischen 53
STRUKTUR 31
SUCC 136
Suchroutine 86
Such-Strings 259
SWAP 254
Symbol 25, 28
Syntax 25
Syntax-Diagramm 25f.
Syntaxfehler 274
Syntaxprüfung 273
Syntaxregeln 34f.
Systemdiskette 14, 16
Systemfile 15
Systemprogrammierung 293

T

TABELLE 38
Tabulator 248
TABULATOR 252
Tagfield 161
Tastaturbelegung 263
Tastatureingabe 17
TEXT 253
Textblock einfügen 268
Textfile 175
–, Ein-/Ausgabe 176
–, READ(LN) 176, 179f.
–, WRITE(LN) 176
Textgrenze 248
Textmodus 258
THEN 59
TOP 248
Tortendiagramm 131
Typ 104, 116
Typkompatibilität 106, 139, 287
Typumwandlung 49

U

Überschreiben 213
Unterbereichstyp 139, 141
UNTIL 70
UPPER 253

V

Variable 31ff.
Variablenparameter 114
Variablenspeicher 239
Variablentyp 32
Verschieben 257
VISIBILITY 152

W

Wertparameter 113
WHILE 64ff., 68
Wiederholungsfaktor 256
WITH 150, 289
Wortsymbole 296
WRITE 35, 38, 91, 304
WRITE(LN) 38, 57, 176, 304

X

XOR 234

Z

Zahlen 26
–, ganze 27, 44
–, reelle 44, 48, 133, 305f.
–, Typen 27
Zahlenkonvertierung 310
Zeichen 48
Zeigervariable 190
Zeilenstruktur 177
Zeilenwechsel 38
Zentrieren 41

Bücher zum Commodore 64/128

S. Vilsmeier
3D-Konstruktion mit GIGA-CAD Plus auf dem C64/C128
1986, 370 Seiten, inkl. 2 Disk.
Mit GIGA-CAD können Computergrafiken von besonderer Räumlichkeit und Faszination geschaffen werden. GIGA-CAD Plus ist schneller und einfacher zu bedienen, die Benutzeroberfläche wurde verbessert und der Befehlssatz erweitert. Die Eingabe erfolgt in erster Linie über den Joystick. Hardware-Anforderung: C64 mit Floppy 1541 oder C128 (im 64'er-Modus), Fernseher oder Monitor, Joystick und Commodore- oder Epson-kompatibler Drucker.
● Das verbesserte GIGA-CAD-Programm mit neuen Features wie erweitertem Befehlssatz und bis zu 10mal schneller liegt dem Buch im Floppy-1541-Format bei.
Best.-Nr. 90409
ISBN 3-89090-409-2
DM 49,–
(sFr 45,10/öS 382,20)

H. Haberl
Mini-CAD mit Hi-Eddi plus auf dem C64/C128
1986, 230 Seiten, inkl. Diskette
Auf der beiliegenden Diskette findet der Leser das vollständige Zeichenprogramm »Hi-Eddi«, mit dem das komfortable Erstellen von technischen Zeichnungen, Plänen oder Diagrammen ebenso möglich ist wie das Malen von farbigen Bildern, Entwurf und Ausdruck von Glückwunschkarten, Schildern, ja sogar von bewegten Sequenzen (kleine Trickfilme, Schaufenster-Werbung).
● Wer sagt, daß CAD auf dem C64 nicht möglich ist?!
Best.-Nr. 90136
ISBN 3-89090-136-0
DM 48,–
(sFr 44,20/öS 374,40)

B. Bornemann-Jeske
Vizawrite-Buch für den C64/C128
1987, 228 Seiten
Mit dem »Vizawrite-Buch« liegt erstmals ein vollständiges und detailliertes Arbeitsbuch für den Anfänger und den professionellen Anwender zur Textverarbeitung auf dem C64/C128 vor. Die Grundlagenkapitel führen Sie anhand kurzer Übungsaufgaben in die elementaren Funktionen des Systems ein. Das Kapitel für Fortgeschrittene zeigt Ihnen jede Programmfunktion im Detail. Zahlreiche praktische Tips aus verschiedenen Anwendungsbereichen ermöglichen Ihnen die optimale Nutzung Ihres Textverarbeitungssystems.
Best.-Nr. 90231
ISBN 3-89090-231-6
DM 49,–
(sFr 45,10/öS 382,20)

O. Hartwig
Experimente zur Künstlichen Intelligenz mit C64/C128
1987, 248 Seiten
Sind Maschinen intelligent? Können Computer denken? Erschließen Sie sich eines der interessantesten Gebiete der modernen Computerforschung! Anhand zahlreicher Programme erfahren Sie hier die Möglichkeiten der Künstlichen Intelligenz, speziell auf dem C64 und dem C128. Der Schwerpunkt des Buches liegt auf der Praxis. Alle KI-Techniken werden durch anschauliche Programme vorgestellt, die sofort nachvollziehbar sind. Zusätzlich erhalten Sie jede Menge Anregungen zu eigenen Experimenten. Die KI-Programme können ohne weiteres in eigene Programme integriert werden.
Best.-Nr. 90472
ISBN 3-89090-472-6
DM 49,–
(sFr 45,10/öS 382,20)

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

706358

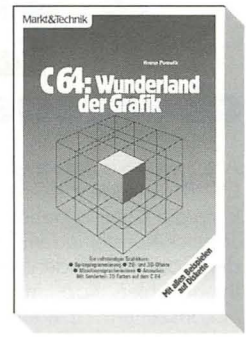
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher zum Commodore 64/128

M. Hegenbarth/R. Trierscheid
BASIC-Grundkurs mit dem C64
1985, 377 Seiten
Kein rein theoretisch ausgelegter BASIC-Kurs, sondern praxisnah auf den C64 zugeschnitten. Auch der Computerneuling kann mit diesem Buch lernen, mit seinem C64 in BASIC zu arbeiten, und wird auf die Besonderheiten seines Computers hingewiesen. Der leichtverständliche, lockere Stil und die gute logische Gliederung der Kapitel unterstützen dies. Erwähnenswert ist ein Kapitel, das die Kommunikation zweier C64 beschreibt, der Anhang, in dem eine Liste nützlicher PEEKs, POKEs und SYS und noch vieles mehr enthalten ist.
● Für den Lesertyp, der beim Lernen auch noch Spaß haben möchte.
Best.-Nr. 90361
ISBN 3-89090-361-4
DM 44,–
(sFr 40,50/öS 343,20)

F. Matthes
Pascal mit dem C64
1986, 215 Seiten, inkl. Diskette
Buch und Compiler ermöglichen jedem Besitzer eines C64 den Einstieg in die moderne Programmiersprache Pascal. Der Compiler akzeptiert den gesamten Sprachumfang mit einigen Erweiterungen. Er bildet mit einem sehr komfortablen Full-Screen-Editor eine schnelle Einheit, so daß der Programmentwicklungsaufwand minimal ist. Übersetzte Programme laufen ohne weitere Hilfsprogramme auf jedem C64, nutzen den gesamten Programmspeicher des C64 und sind 3-4mal schneller als vergleichbare Programme in BASIC. Dem Buch liegt ein leistungsfähiges Pascal-System mit einigen Pascal-Programmen auf Diskette bei.
Best.-Nr. 90222
ISBN 3-89090-222-7
DM 52,–
(sFr 47,80/öS 405,60)

W. Kassera/F. Kassera
C64-Programmieren in Maschinensprache
Der Aufschwung im Programmieren stellt sich ein, wenn Sie die betriebssysteminternen ROM-Routinen kennen, über ihre Funktionsweise und ihr Zusammenspiel informiert sind. Und Sie müssen die Maschinensprache Ihres C64 beherrschen. Beides ermöglicht Ihnen dieses Buch. Es zeigt, wie Sie bewegte Bildschirmobjekte programmieren, die Interrupt-Routine des Systems erweitern, die Arithmetik-Routinen im ROM und deren Datentypen beherrschen, und alles, was Sie über Ein-/Ausgabe, BASIC-Variable und andere wichtige Themen wissen müssen.
Best.-Nr. 90168
ISBN 3-89090-168-9
DM 52,–
(sFr 47,80/öS 405,60)



H. Ponnath
C64: Wunderland der Grafik
1985, 232 Seiten, inkl. Diskette
Der Autor legt beim Leser ein solides Fundament an Wissen, und er tut dies auf so unterhaltsame Art, daß Sie bestens gerüstet sind, um so interessante Aufgaben wie die Programmierung hochauflösender zwei- und dreidimensionaler Grafiken anzugehen. Mit Sprites zu jonglieren ist für Sie bald kein Problem mehr, aber auch das vertrackte Verdeckungsproblem bei dreidimensionaler Grafik kriegen Sie jetzt endlich in den Griff. Finden Sie heraus, was wirklich im Grafik-Chip Ihres C64 steckt!
● Eine lesenswerte und kenntnisreiche Einführung in dieses hochinteressante Thema von einem sachkundigen Autoren; mit allen Beispielen auf beigefügter Diskette.
Best.-Nr. 90363
ISBN 3-89090-363-0
DM 49,–
(sFr 45,10/öS 382,20)

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

706359

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

64'er C= Spiele sammlung

Lassen Sie sich in eine abenteuerliche Spielewelt entführen!

Alles, was Sie brauchen, ist ein C64 oder ein C128, beiliegende Spielediskette – und schon kann die Reise losgehen. Beweisen Sie Ihre Joystick-Künste, indem Sie sicher den Weg aus dem Labyrinth finden! Bewahren Sie Ihren kühlen Kopf in aufregenden Actionszenen! Zeigen Sie Ihre Fähigkeiten als Börsenmakler in lebensnahen Wirtschaftssimulationen! Mit den 15 spannenden Spielen, der ausführlichen Anleitung sowie den farbigen Bildschirmfotos ist Ihnen ein fantastisches Spielvergnügen gewiß.

Aus dem Inhalt:

Balliard: Einfallswinkel = Ausfallswinkel. Wer das nicht befolgt, hat es schwer bei dieser Mischung aus Tennis und Billard.

The Way: Zu verschlungenen Pfaden gesellen sich Geldsäcke und böse Geister, die es zu bekämpfen gilt. **Vager 3**: Joystickprofis mit ungetrübtem Visierblick und Trefferinstinkt können ihr Punktekonto schwer mit Abschußprämien beladen.

Firebug: Hoffentlich fängt Ihr Joystick nicht ebenfalls Feuer, wenn es heißt, die wertvollen Koffer aus dem brennenden Haus des Professors zu erwischen. **Pirat**: Taktik, Timing und gute Navigationskenntnisse sind Voraussetzung für ein bis zu 25 Jahre langes Piratenleben. **Wirtschaftsmanager**: Simulation aus den höchsten Etagen der Wirtschaft, nicht 1000 Stück, sondern ganze Firmen gehen über den »Ladentisch«. **Vier gewinnt**: Einfach, aber gerade deshalb ein Spiel, das schnell zu Erfolgserlebnissen führt. **Brainstorm**: Mastermind stand Pate für dieses vielseitige Denkspiel. **Hypra-Chess**: Spielen Sie Schach gegen einen C64 und außerdem die Spiele **Maze, Schiffe versenken, Handel, Börse, Vier in vier und Magic-Cubs.**

Hardware-Anforderungen: C64 oder C128 bzw. C128D (64er-Modus), Floppy 1541, 1570 oder 1571 und Joystick.

Best.-Nr. 90429, ISBN 3-89090-429-7

DM 39,–* **(sFr 35,90/öS 304,20)**

*inkl. MwSt. Unverbindliche Preisempfehlung

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, Computerfachhändler oder in den Fachabteilungen der Warenhäuser. Fragen Sie auch nach dem neuen Gesamtverzeichnis Herbst/Winter '87/88.

708125/2

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bitte schneiden Sie diesen Coupon aus, und schicken Sie ihn in einem Kuvert an:
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar

Computerliteratur und Software vom Spezialisten

Vom Einsteigerbuch für den Heim- oder Personalcomputer-Neuling über professionelle Programmierhandbücher bis hin zum Elektronikbuch bieten wir Ihnen interessante und topaktuelle Titel für

• Apple-Computer • Atari-Computer • Commodore 64/128/16/116/Plus 4 • Schneider-Computer • IBM-PC, XT und Kompatible
sowie zu den Fachbereichen Programmiersprachen • Betriebssysteme (CP/M, MS-DOS, Unix, Z80) • Textverarbeitung • Datenbanksysteme • Tabellenkalkulation • Integrierte Software • Mikroprozessoren • Schulungen.
Außerdem finden Sie professionelle Spitzen-Programme in unserem preiswerten Software-Angebot für Amiga, Atari ST, Commodore 128, 128 D, 64, 16, für Schneider-Computer und für IBM-PCs und Kompatible!
Fordern Sie mit dem nebenstehenden Coupon unser neuestes Gesamtverzeichnis und unsere Programmservice-Übersichten an, mit hilfreichen Utilities, professionellen Anwendungen oder packenden Computerspielen!

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2,
8013 Haar bei München, Telefon (089) 4613-0

Adresse:

Name

Straße

Ort

Bitte schicken Sie mir:
☐ Ihr neuestes Gesamtverzeichnis
☐ Eine Übersicht Ihres Programmservice-Angebotes aus der Zeitschrift

☐ Außerdem interessiere ich mich für folgende/n Computer:

(PS: Wir speichern Ihre Daten und verpflichten uns zur Einhaltung des Bundesdatenschutzgesetzes)

Markt&Technik Verlag AG
– Unternehmensbereich Buchverlag –
Hans-Pinsel-Straße 2
D-8013 Haar bei München

Die 64'er-Langspiel-Diskette

ACHTUNG!
Computer-Freaks aufgepaßt:
32 Spitzen-Musikprogramme aus dem 64'er-Musik-Programmierwettbewerb auf einer Diskette mit komfortablem Lademenü. Von Pop bis Klassik ist für jeden Musikgeschmack etwas dabei: Shades, This is not America, Invention Nr. 13, Mondscheinsonate, You can win if you want, Der Clou, Für Elise, The pink Panther und viele mehr.

Hardware-Anforderungen:
Commodore 64 oder Commodore 128 im C-64-Modus, Floppy-Station 1541, 1570 oder 1571

Ein »Muß« für jeden 64'er-Fan!

Best.-Nr. 39630
DM 39,90*
(sFr 34,90/öS 399,-*)
*inkl. MwSt. Unverbindliche Preisempfehlung

Einmalig in der Computergeschichte:
- Alle Musikstücke werden in Stereoqualität auf einer hochwertigen Kassette mit Rauschunterdrückung mitgeliefert!
- Eineinhalb Stunden erstklassige Computermusik!
- Klang umwerfend!

Lieferumfang:
1 Diskette beidseitig bespielt mit 32 Musikstücken
1 Kassette mit allen Musikstücken in Stereoqualität für handelsübliche Kassettenrecorder oder Stereoanlagen

Markt&Technik-Softwareprodukte erhalten Sie in den Fachabteilungen der Kaufhäuser, in Computershops oder im Buchhandel.

Markt&Technik
Zeitschriften · Bücher ·
Software · Schulung

706229-2

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

FLORIAN MATTHES,
geboren 1963 in Frankfurt am Main. Abitur 1981. Ab 1983 Studium der Informatik mit Nebenfach Physik an der Universität Frankfurt. Seit 1985 wissenschaftlicher Mitarbeiter am Lehrstuhl für Technische Informatik. Einige Veröffentlichungen in Mikrocomputer-Fachzeitschriften und freiberufliche Arbeit in der DV-Branche.

Pascal mit dem C128

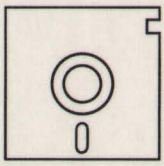
Dem Buch liegt auf Diskette ein leistungsfähiges Pascal-System mit zahlreichen Beispiel- und Übungsprogrammen bei. Buch und Compiler ermöglichen jedem Besitzer eines C 128 oder C 128 D den Einstieg in die moderne Programmiersprache Pascal.
Der Anfänger findet im Buch einen vollständigen praxisorientierten Einführungskurs in Pascal, wobei viele überschaubare Beispiele und interessante Aufgaben zum aktiven Lernen am Computer auffordern. Beim Programmieren wird er durch eine ausführliche Bedienungsanleitung des Systems unterstützt.
Dem fortgeschrittenen Anwender erlaubt der vollständige Sprachumfang, der um String-, Grafik- und systemnahe Prozeduren erweitert ist, die Entwicklung komplexer Programme, die alle Fähigkeiten des C128 ausnutzen.
Der Compiler bildet mit einem sehr komfortablen Fullscreen-Editor eine leicht zu bedienende und schnelle Einheit, so daß der Aufwand zur Programmentwicklung minimal ist. Übersetzte Programme nutzen den gesamten vorhandenen Speicher und sind drei- bis zehnmal schneller als vergleichbare Programme in BASIC.
Aus dem Inhalt:

- Leistungsfähiger Compiler mit Editor auf Diskette
- Vollständiger Einführungskurs in Pascal
- Beispiele und Aufgaben
- Tips und Tricks für den Profi
- Ausführliche Bedienungsanleitung

Hardware-Anforderungen:
C 128 mit Floppy 1541, 1570, 1571 oder C 128 D

ISB N 3-89090-386-X

Markt&Technik



DM 52,-
sFr 47,80
öS 405,60